Edge Computing on the IoT Gateway

Francisco Javier Pacheco Herranz

MÁSTER EN INTERNET DE LAS COSAS. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Internet de las Cosas

Curso 2018/2019
Convocatoria Septiembre de 2019
Calificación: 9.5 (Matrícula de Honor)

Directores:

Luis Piñuel Moreno
Francisco Daniel Igual Peña

Autorización de difusión

FRANCISCO JAVIER PACHECO HERRANZ

3 de septiembre de 2019

El/la abajo firmante, matriculado/a en el Máster en Internet de las Cosas de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Edge Computing on the IoT Gateway", realizado durante el curso académico 2018-2019 bajo la dirección de Luis Piñuel Moreno y Francisco Daniel Igual Peña en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

Las arquitecturas *cloud* tradicionales se van a enfrentar, debido a la proliferación de dispositivos IoT, a una serie de dificultades en términos de latencia, ancho de banda y seguridad.

El *Edge Computing* se presenta como una tecnología que pretende solucionar estos problemas acercando la capacidad de cómputo al borde de la red. Para ello, el principal mecanismo es situar *gateways* con capacidad de cómputo en localizaciones cercanas a los sensores. Estos dispositivos realizarán inferencias sobre los datos generados para reducir el volumen de información enviada a la nube.

En este trabajo se introduce ECM (*Edge Computing Manager*), un entorno desarrollado para actuar como la capa de lógica dentro de un dispositivo de borde situado en el contexto de una casa conectada. Prestará servicios levantados bajo demanda, que proveerán a los usuarios de información recogida por los sensores que tengan situados a lo largo de una casa.

Se probará que como dispositivo de borde se puede utilizar un dispositivo de baja prestaciones, gracias a servicios de virtualización ligera como *Docker*.

Palabras clave

Internet de las Cosas, Edge Computing, Enrutador, Dispositivo de borde, Docker, Nube, Análisis de datos, Contenedor

Abstract

Traditional cloud architectures will face several problems due to the proliferation of IoT devices. This problems are mainly related to latency, bandwidth consumption and security.

Edge Computing emerges as a technology that pretends to tackle this problem, by bringing computational capacities closer to the edge of the network. The main mechanism to achieve this goal is the deployment of gateways close to the sensors. The edge device will perform inference from data generated by sensors, so the amount of information sent to the cloud decreases.

In this work we introduce ECM (*Edge Computing Manager*), a software environment developed to perform as the logical layer of an edge device, deployed in the context of house automation. It will run on-demand services which will provide users with information obtained from the sensors situated along the house.

It will be proved that a low power device is suitable to act as an edge device, thanks to lightweight virtualization services such as Docker.

Keywords

Internet of things, Edge Computing, Gateway, Edge device, Docker, Cloud, Data analysis, Container

Índice general

Índice I

Agradecimientos III

Dedicatoria IV

1. Introducción **1**
1.1. Motivación 1
1.2. Objetivos 3
1.3. Método de trabajo 5
1.4. Organización del documento 7

2. Estado del arte **8**
2.1. Iniciativas *Edge Computing* 8
2.1.1. *Cloudlet* 8
2.1.2. *Fog Computing* 10
2.1.3. Iniciativa *Multi-access Edge Computing* 11
2.1.4. Edge Computing en grandes proveedores de servicios cloud 12
2.2. Dispositivos de borde 15
2.3. Tecnologías de virtualización y contenerización de servicios 17
2.3.1. Docker como servicio de virtualización ligero 18

3. ECM. Diseño y funcionalidad **24**
3.1. Registro de cliente 24
3.2. Levantamiento de servicio 25
3.2.1. Servicio de escucha de datos 25
3.2.2. Servicio de análisis de datos 27
3.2.3. Servicio personalizado 28
3.2.4. Parámetros adicionales 29
3.3. Consultar servicios activos 29
3.4. Detener servicio 29
3.5. Eliminar servicio 30
3.6. Eliminar perfil 31

4. ECM. Desarrollo **32**
4.1. Casos de uso 32
4.2. Arquitectura de la solución 34
4.2.1. Diagrama de despliegue del sistema 34

4.2.2. Diseño del servicio . . . 35
4.3. Diagramas de secuencia . . . 38
4.4. Gestión de recursos . . . 44
4.4.1. Solicitud de servicio . . . 44
4.4.2. Eliminación de servicio . . . 45
4.5. Modelado y construcción de servicios . . . 46
4.5.1. Construcción de los contenedores . . . 46
4.5.2. Diseño de los servicios . . . 47

5. ECM. Evaluación de rendimiento **57**
5.1. Valoración del funcionamiento del sistema . . . 57
5.1.1. Rendimiento de análisis en modo *batch* . . . 58
5.1.2. Rendimiento de análisis en modo dato a dato . . . 62
5.1.3. Tiempo total de provisión de servicio . . . 67
5.1.4. Capacidad máxima de la plataforma . . . 72
5.1.5. Aislamiento de contenedores . . . 74

6. Conclusiones **80**

7. Introduction **82**
7.1. Motivation . . . 82
7.2. Objectives . . . 84
7.3. Work methodology . . . 85
7.4. Document organization . . . 87

8. Conclusions **88**

Bibliography **91**

Agradecimientos

Quiero agradecer enormemente a todas las personas que, directa o indirectamente, me han ayudado a alcanzar esta meta. Ya fuese con unas palabras de ánimo o con un apoyo constante, han hecho que el camino haya resultado más sencillo.

Especialmente quiero agradecer a mis padres y a mi pareja su incondicional apoyo.

También quiero agradecer a mis tutores, los doctores Francisco D. Igual Peña y Luis Piñuel Moreno, la confianza depositada en mí cuando me ofrecieron la posibilidad de realizar este trabajo, y su indispensable ayuda a lo largo de todo su desarrollo.

Dedicatoria

A todos aquellos que confiaron en mí, y me soportaron en todo momento, gracias de corazón.

Capítulo 1

Introducción

1.1. Motivación

Internet de las Cosas (IoT) está evolucionando de tal manera que, para 2024, se calcula que habrá 4000 millones de dispositivos conectados a Internet únicamente en entornos IoT. Si a esa ya de por sí enorme cantidad de dispositivos conectados le añadimos otros tantos tales como teléfonos móviles, la cifra prevista asciende hasta los 8900 millones [5].

Esta situación amenaza las arquitecturas *cloud* tradicionales, que hasta el momento presentan un modelo centralizado en la que un único centro de datos aglutina gran cantidad de tráfico proveniente de múltiples sistemas conectados [22, 5, 21]. Este modelo presenta una serie de desventajas que iniciativas como *Edge Computing* pretenden solucionar.

El primer problema es el aumento de la *latencia*. Las aplicaciones IoT pueden requerir tasas de latencia muy bajas. En caso contrario, la calidad de servicio (QoS, Quality-of-Service) y la calidad de la experiencia (QoE, Quality-of-Experience) pueden verse degradadas [16]. Además, no solo el incremento de dispositivos conectados afecta a esta restricción: la distancia entre estos dispositivos y los centros de datos, actualmente, es lo suficientemente grande como para resultar perjudicial a las comunicaciones [22, 16].

El segundo problema es el consumo de *ancho de banda*. En algunos contextos, el volumen de los datos generados por los dispositivos IoT puede llegar a ser, en cuestión de segundos, del orden de Gigabytes. Esto supone que el ancho de banda existente en las comunicaciones

actuales limite las capacidades de computación, sobretodo aquellas que requieran llevarse a cabo en tiempo real [26]. En resumen, no solo ocurre que la computación no sea tan inmediata como es deseable; la cantidad ingente de datos ocasiona congestión en la red [21, 18, 22].

El tercer problema está relacionado con la *seguridad* [10, 22]. Por un lado, todo dato enviado a la nube se ve expuesto a posibles ataques. Los proveedores de servicios *cloud* tienen que estar continuamente evolucionando y mejorando sus soluciones de seguridad, dado que un despliegue IoT puede realizarse en contextos muy distintos que requieran sistemas de seguridad muy distintos, como pueden ser entornos industriales o domésticos [4]. Además, puede entenderse como seguridad la capacidad que tiene un sistema de resistir caídas del sistema. Algunos autores apuntan que, cuanto más centralizado esté un sistema, menos capacidad tiene para sobreponerse a ese tipo de situaciones [21, 18].

Ante estos problemas, *Edge Computing* emerge como un paradigma que trata de mejorar la eficiencia global de la infraestructura de los despliegues IoT [21, 18, 19, 20].

Para ello, persigue el objetivo de acercar al borde de la red capacidad de cómputo, almacenamiento y *networking* [26, 18]. Las arquitecturas características de esta tecnología pasan por desplegar el *gateway* en el *edge*, que puede considerarse cualquier recurso de cómputo o red situado en el camino entre la fuente de datos y los centros de datos [26]. Estas localizaciones se encuentra relativamente cercanas a los sensores. Además de ejecutar las tareas propias de un *gateway* -como el *fowarding* de datos-, el dispositivo de borde también aporta estas nuevas capacidades.

De esta manera, todos los datos generados por los sensores de un sistema son almacenados y procesados al borde de la red, enviando a la nube únicamente aquellos datos de verdadero interés. Así, al reducir el volumen de datos enviado, las comunicaciones con el servidor externo se ven descongestionadas, reduciendo latencia y consumo de ancho de banda.

Hasta ahora, las arquitecturas *cloud* presentaban técnicas como el *offloading*, que consiste en delegar en la nube operaciones computacionales que no pueden realizarse de manera local

por falta de potencia [24]; y el *caching*, que permite acceder a los datos que se solicitan frecuentemente de manera más rápida, aumentando así la eficiencia de un sistema [6]. Estas técnicas también son aplicables en los dispositivos de borde [22, 26, 18].

Estas utilidades descubren potenciales nuevos servicios e inexplorados modelos de negocio [18]. Industrias como las *smarthome* o los coches autónomos pueden verse ampliamente beneficiados gracias a esta tecnología [26].

Centrándonos en el ámbito de las casas conectadas, se puede observar de manera manifiesta las ventajas que aporta una arquitectura *Edge Computing*.

Los datos generados por sensores en *smarthomes* son de tamaños limitados, como puede ser una medición de temperatura o de humedad. Sin embargo, contando todos los dispositivos que están midiendo parámetros ambientales, y la frecuencia con la que capturan valores, a lo largo del día pueden producir una cantidad de datos considerable.

Estos datos son susceptibles de ser analizados para detectar anomalías y situaciones de riesgo, pero, ¿tiene sentido que sean enviados a la nube, cuando el análisis necesario a realizar sobre ellos resulta relativamente sencillo?

En este sentido, una arquitectura *Edge Computing* resulta de gran utilidad, pues permite realizar inferencias sobre los datos en el propio dispositivo de borde, que pasa a enviar una cantidad de datos mucho menor cada día; por ejemplo, solo el resultado fruto del análisis realizado sobre todos los valores ambientales capturados durante un día [13].

Así, las comunicaciones con la nube disminuyen, reduciendo el cuello de botella que puede llegar a producirse, y también reduciendo las posibilidades de que un dato se vea expuesto a una amenaza en forma de ciberataque.

1.2. Objetivos

El principal objetivo del trabajo es demostrar que es posible desplegar un sistema *Edge Computing* sobre un *gateway* de bajas prestaciones, desarrollando una infraestructura software que añada una capa de inteligencia sobre las que ya presenta un dispositivo de borde

de manera usual: tareas de enrutamiento y *forwarding* de datos.

La solución desarrollada adoptará el nombre de *Edge Computing Manager* (ECM).

Esta arquitectura debe proveer capacidades para realizar análisis de datos en el borde. La idea es realizar inferencias sobre los datos que el *gateway* recibe de manera continua, y desarrollar un software capaz de automatizar todas las tareas de ingesta de datos, análisis y emisión de resultados a partes interesadas.

Todo el ecosistema ha de cumplir una serie de requisitos en cuanto funcionalidad, eficiencia, capacidad de procesamiento y seguridad.

Se construirá sobre un dispositivo de borde un software capaz de levantar servicios bajo demanda con diversas funcionalidades, desde la ingesta de datos hasta su análisis, además de proveer tanto los datos en bruto como los resultados de las inferencias a los clientes.

Se probará que el sistema soporta cargas de trabajo en las que participen distintos clientes, que soliciten datos provenientes de distintas fuentes y que requieran análisis.

Se desarrollará un algoritmo capaz de gestionar el rendimiento del sistema, permitiendo modificar dinámicamente los recursos destinados a los servicios levantados en función de la demanda que exista. Así, en periodos de poca demanda, los servicios levantados disfrutarán de más recursos para realizar las tareas solicitadas; por el contrario, en periodos de mucha demanda, cada servicio contará con recursos más limitados.

También se probará que el sistema es seguro frente a diferentes vulnerabilidades. Por un lado, se asegurará la privacidad de los datos, que estarán únicamente disponibles para el cliente dueño de los mismos, estando los diferentes servicios levantados aislados entre ellos. Por otro lado, se desarrollarán mecanismos para evitar suplantaciones de identidad por parte de los atacantes, evitando que se hagan pasar por clientes registrados en el sistema.

Se realizarán análisis de rendimiento para comprobar que, como se ha indicado, el sistema soporta múltiples solicitudes por parte de diversos clientes, y que puede mantener funcionando al mismo tiempo varios servicios.

Para contextualizar el proyecto, la funcionalidad del *gateway* a desarrollar está pensada

para funcionar, por ejemplo, en un hogar conectado. Esta vivienda dispondrá de varios sensores distribuidos por diferentes habitaciones, y estarán enviando los valores capturados al dispositivo de borde.

La idea es que este dispositivo sea capaz de realizar un pequeño análisis computacional de estos valores, para obtener ciertas métricas que serán las enviadas a la nube. Por tanto, dentro de este *gateway* se desplegarán varios servicios aislados en contenedores Docker [7]; unos servicios escucharán por puertos de entrada para recibir y almacenar los valores enviados por los sensores, mientras que otros servicios ejecutarán tareas de análisis sobre estos datos.

Esta arquitectura puede verse incluida dentro de otra de mayor alcance, como puede apreciarse en la figura 1.1. En un edificio de viviendas, cada dispositivo *edge* de cada apartamento se comunica con un dispositivo común para todos -notablemente más potente-, que procesa los datos enviados para, a su vez, comunicarse con *gateways* superiores. Estos dispositivos pueden procesar los datos de una manzana, o de todo un barrio, hasta eventualmente comunicarse con la nube.

1.3. Método de trabajo

El desarrollo del proyecto seguirá los siguiente pasos:

- Selección del dispositivo de borde a utilizar. El critierio de dicha selección se basa en una serie de premisas: que el consumo del dispositivo sea bajo y que tenga una mínima capacidad de cómputo.

- Desarrollo de pruebas para comprobar que ofrece capacidad de cómputo suficiente para desplegar servicios de análisis de datos. Dada la restricción de que la arquitectura es de 32 bits, comprobar que la instalación de Docker es viable, así como el uso de librerías de análisis de datos.

- Toma de requisitos necesarios para definir la funcionalidad de la solución: cómo ges-

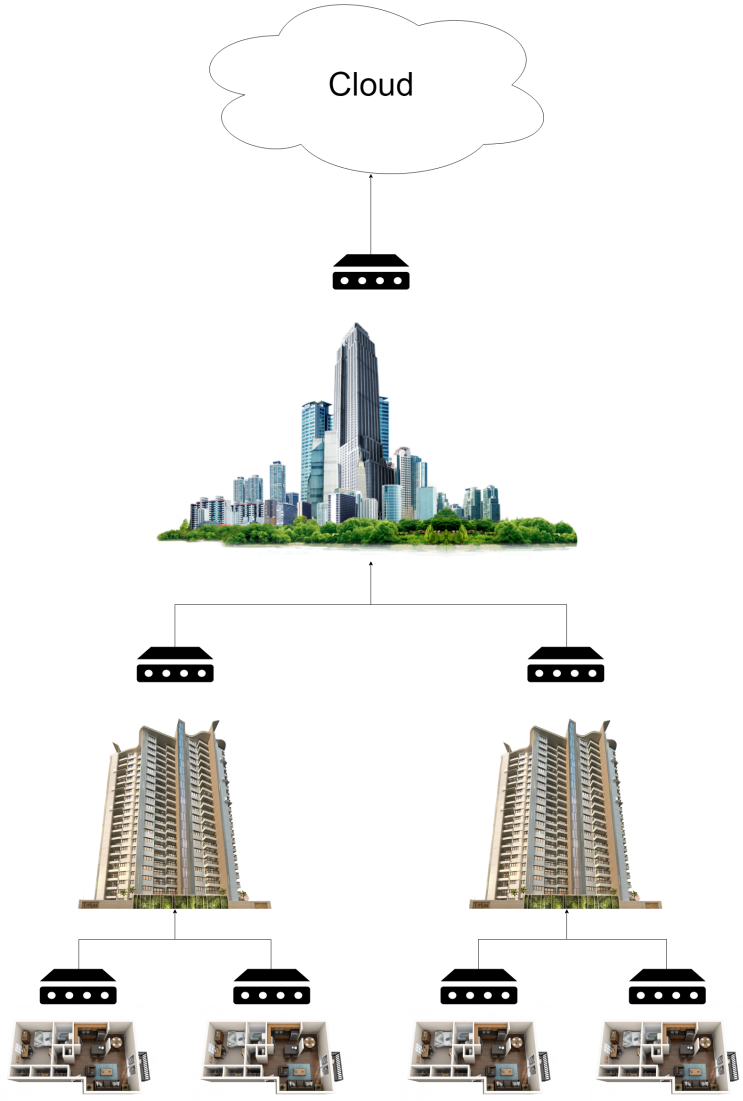


Figura 1.1: Arquitectura del entorno objetivo

tionar los contenedores y su consumo de recursos del sistema, cómo gestionar las conexiones con los clientes, qué tipo de servicios ofrecer, etc.

- Construicción de la solución. En primera instancia, desplegar sistemas sencillos, que vayan siendo evolucionados hasta conformar la plataforma final.

- Realizar una serie de pruebas para comprobar, por un lado, que el sistema cumple con los objetivos definidos y, por otro lado, para medir el rendimiento del mismo.

1.4. Organización del documento

El documento se compone de ocho capítulos descritos a continuación.

Este primer capítulo sirve a modo de introducción, planteando la idea del proyecto y los objetivos a alcanzar.

En el segundo capítulo se introduce el estado del arte. Aquí se hablará, primeramente, acerca de qué proyectos, iniciativas y plataformas *Edge Computing* existen en la actualidad, mostrando las arquitecturas definidas y las tecnologías utilizadas. Después se profundizará en los dispositivos candidatos a ser dispositivo de borde, y qué tecnologías se utilizarán para desarrollar la solución.

En el tercer capitulo se describe la funcionalidad de la plataforma. Se presentarán los servicios proporcionados, cómo el cliente interactúa con estos y qué resultado se espera obtener de cada uno.

En el cuarto capítulo se describe el desarrollo de la plataforma. Se presentarán una serie de diagramas UML, que representarán la arquitectura de la solución mediante diagramas de despliegues y objetos, y el funcionamiento de todos los servicios mediante diagramas de secuencia. También se ilustrarán ciertos componentes de manera más detallada.

En el quinto capítulo se documentarán todas las pruebas realizadas sobre la plataforma. Se incluirán tablas y gráficas que ilustrarán los resultados obtenidos, y qué conclusiones se pueden obtener de dichos análisis.

En el sexto capítulo se presentarán las conclusiones del proyecto y el cumplimiento de los objetivos propuestos.

En el séptimo y octavo capítulo se presentarán la introducción y las conclusiones traducidas al inglés.

Capítulo 2

Estado del arte

A continuación va a describirse una imagen general que mostrará en qué estado se encuentra la tecnología *Edge Computing*.

Por un lado, se expondrán los proyectos e iniciativas más importantes que se estén llevando a cabo cuyas arquitecturas reúnan las condiciones necesarias para ser consideradas como arquitecturas *Edge Computing*. Se describirán estas propuestas, señalando qué tecnologías utilizan y cómo están construidas las soluciones.

Por otro lado, se presentarán distintos dispositivos hardware que puedan ser utilizados como *gateway* o dispositivo de borde, comparando sus prestaciones y especificaciones técnicas

Además, se enumerarán y describirán distintas tecnologías software adecuadas para construir una arquitectura *Edge Computing*, qué ventajas e incovenientes presenta cada una de ellas y cuál es la apropiada para utilizar en el contexto de este proyecto.

2.1. Iniciativas *Edge Computing*

2.1.1. *Cloudlet*

Cloudlet es un proyecto que surge en la Universidad Carnegie Mellon y que persigue converger el Internet de las Cosas junto con la computación en la nube [14]. Para ello, se postula como una capa intermedia en una arquitectura de tres capas, como puede apreciarse

en la figura 2.1.

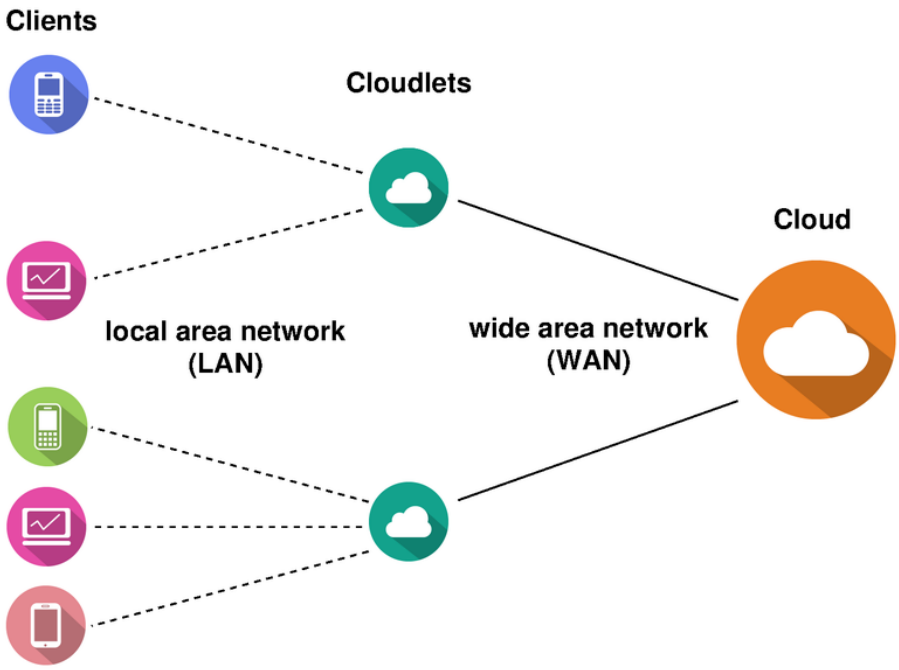


Figura 2.1: Arquitectura Cloudlet

Esta arquitectura es capaz de lidiar con las limitaciones expuestas anteriormente, en relación con el ancho de banda y la latencia. Por un lado, acercar los recursos de computación a los dispositivos móviles disminuye la latencia, pues la distancia entre estos es reducida. Por otro lado, al estrucutarse en una red de área local (*Local Area Network*, LAN), el ancho de banda es mayor.

Por dentro, un *Cloudlet* se basa en el uso de máquinas virtuales que son desplegadas cada vez que un dispositivo se conecta. La infraestructura *Cloudlet* cuenta con una imagen base de la máquina virtual, y es el dispositivo cliente el que aporta un recubrimiento que termina de definir las características de esta. Cuando la conexión con el dispositivo finaliza, la máquina virtual es eliminada dinámicamente sin afectar al resto de dispositivos conectados en la red. Esta técnica es conocida como *dynamic VM synthesis* [25].

2.1.2. *Fog Computing*

La tecnología *Fog Computing* nace con el objetivo de satisfacer los requerimientos que demandan las nuevas aplicaciones sensibles a altas latencias, que requieren la presencia de nodos a una distancia menor que a la que se encuentran, típicamente, los grandes centros de procesamiento de datos [3].

Pretende proveer servicios de computación, almacenamiento y *networking* entre los dispositivos y la nube. Ha de satisfacer las necesidades de aplicaciones que requieran tanto analítica en tiempo real como incluso analítica Big Data, y que puedan actuar en un rango amplio de espacio. Esto conlleva a que los nodos deban estar ampliamente distribuidos, y que sean capaces de operar entre ellos; aparte de su inherente capacidad de conectar con servicios en la nube.

Por ejemplo, para desplegar un sistema que controle los semáforos y las señales en una ciudad, los nodos distribuidos a lo largo de una prolongada avenida han de poder comunicarse para sincronizar correctamente los semáforos. Además, pueden servir como radares; si dos detectores detectan una matrícula en un lapso de tiempo menor al que correspondería si se hubiese completado la distancia que los separa a una velocidad correcta, se considera que dicho vehículo ha superado el límite de velocidad. Este sistema también ha de ofrecer analítica de datos a largo plazo, recibiendo una gran cantidad de información que ha de enviar a la nube. Así pueden estimarse los usuarios diarios y demás tipo de parámetros de interés para realizar tareas de predicción del uso de dicha avenida.

Así, el *Fog Computing* se compone en cuatro capas. La más cercana a los dispositivos ofrece comunicaciones máquina a máquina (M2M, *Machine-to-machine*), para la ingesta de los datos recogidos por los sensores, su procesamiento, filtrado e incluso activación de dispositivos actuadores. La segunda y tercera capa se encargan de la visualicación y reporte de estos datos mediante una interacción humano a máquina (HMI, *Human-to-machine*). Esto provoca que un sistema *Fog* debe ser capaz de almacenar datos efímeros en sus capas más bajas, y otros datos semi permanentes en las capas más altas. Es necesario precisar que,

cuanto más alta sea la capa, mayor es la región geográfica que cubre, y mayor es el tiempo que tarda en realizar el procesamiento de los datos.

En la última capa se situaría la nube, que es la encargada de desplegar una serie de paneles de control que muestren todos los datos que van siendo sensorizados. Toda esta arquitectura puede apreciarse en la figura 2.2.

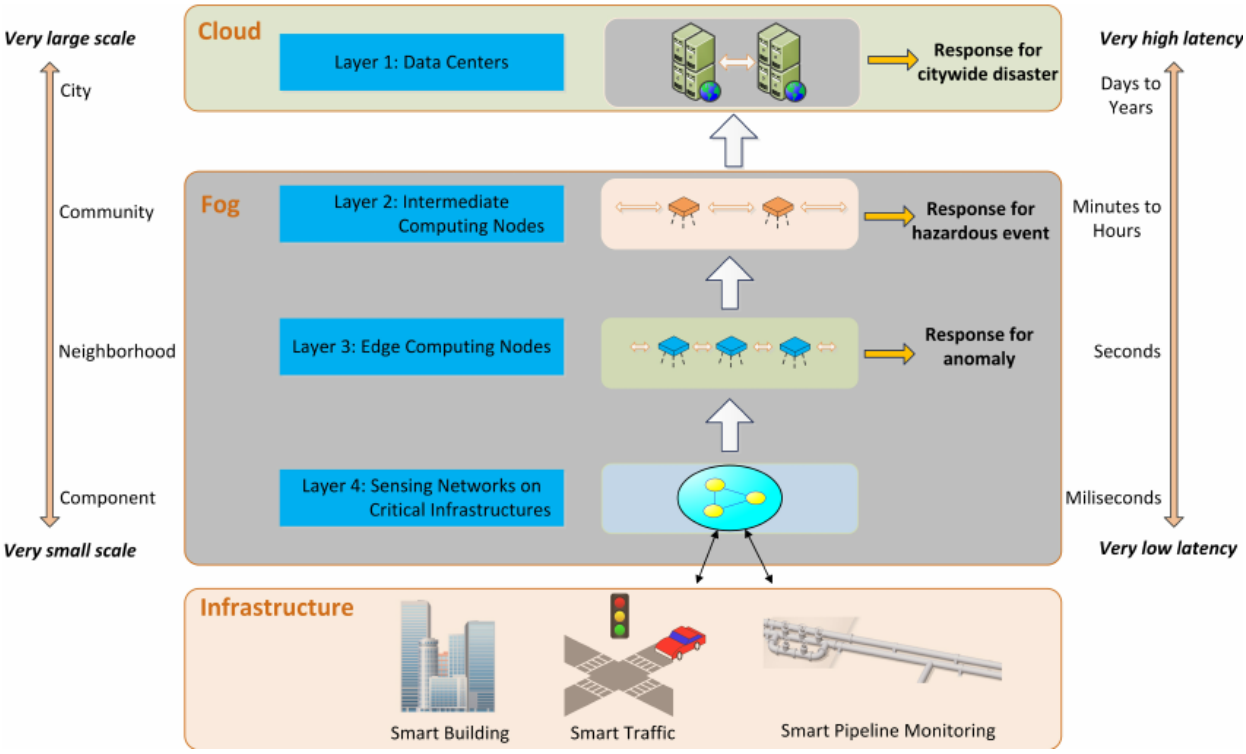


Figura 2.2: Arquitectura Fog Computing en una Smart City

2.1.3. Iniciativa *Multi-access Edge Computing*

El Instituto Europeo de Normas de Telecomunicaciones (ETSI, *European Telecommunications Standards Institute*) lanzó, a través de su grupo técnico *Industry Specification Group* (ISG), la arquitectura *Multi-access Edge Computing* (MEC).

Esta tecnología pretende acercar al borde de la red capacidades y servicios tecnológicos de la nube, asegurando así un gran ancho de banda y una latencia muy baja. Está orientada a casos de uso que requieran interacciones en tiempo real con el usuario, como servicios de

localización, realidad aumentada, *data caching*, analítica de vídeos e IoT.

Su principal objetivo es aprovechar las capacidades del 5G para ofrecer esta calidad de servicio, aunque de momento se sustenta en redes 4G/LTE [17]. Gracias a esto puede ofrecer tres tipos de *Point-of-presence* (PoP): Cliente, cerca del servidor y lejos del servidor. La localización cliente puede ser un teléfono móvil.

Un aspecto negativo de esta arquitectura es que todo desarrollo software ha de adaptarse a sus especificaciones. Para ello, ofrece una API bastante estandarizada con el objetivo de facilitar, en la medida de lo posible, la programación de estas aplicaciones [23].

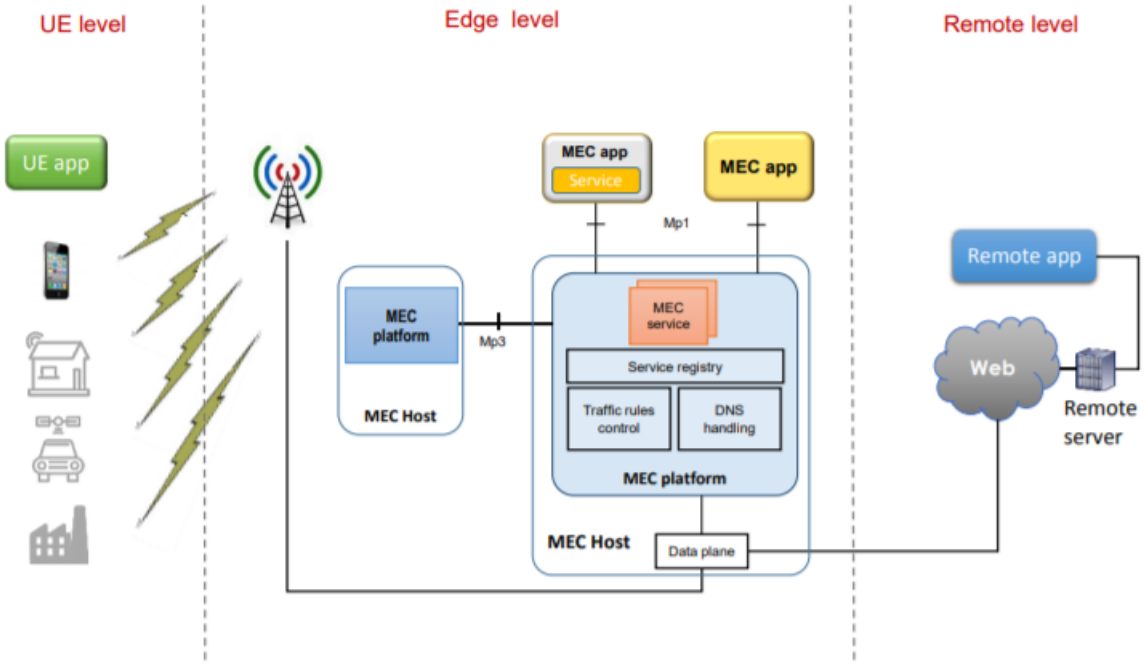


Figura 2.3: Arquitectura Multi-access Edge Computing

2.1.4. Edge Computing en grandes proveedores de servicios cloud

Las grandes empresas tecnológicas también han lanzado iniciativas y servicios de *Edge Computing*. A continuación van a describirse los servicios provistos por dos de los grandes proveedores de servicios en la nube, Microsoft Azure [2] y Amazon Web Services [1].

Azure IoT Edge

La solución de Microsoft pasa por implementar localmente los servicios ya existentes en Azure, tanto a nivel de analítica como para aplicar lógica de negocio.

Se basa en su solución PaaS para el Internet de las Cosas, llamada *IoT Hub*, que ofrece un servicio de aprovisionamiento, ingesta de datos, almacenamiento, análisis y aplicación de lógica de negocio.

Está formado por tres componentes:

- Módulos de IoT Edge: Ejecutan servicios de Azure, de terceros o código propio. Estas unidades de ejecución se implementan como contenedores compatibles con Docker. Estos módulos pueden ser configurados para que se comuniquen entre ellos, e incluso pueden ser ejecutados sin conexión.

- Entorno de tiempo de ejecución: Se encuentra dentro del propio dispositivo IoT Edge. Este servicio está orientado a tareas de administración y comunicación, como el mantenimiento del dispositivo, gestión de actualizaciones, mantenimiento de los estándares de seguridad y la garantía de que cada módulo está siempre en ejecución.

- Interfaz en la nube: El sistema entero puede ser administrado a través de una Interfaz en la nube. Permite crear y configurar cargas de trabajo, enviarlas a conjuntos de dispositivos y supervisarlas.

Amazon *IoT Greengrass*

La solución propuesta por Amazon Web Services (AWS) sigue la misma filosofía que *Azure IoT Edge*: acercar a los dispositivos de borde capacidad de cómputo y administración.

Es una solución que permite ejecutar varios de los servicios de AWS en los dispositivos conectados. Provee de un servicio que permite realizar un filtrado de los mensajes que son enviados a la nube, para así transmitir únicamente la información necesaria. También ofrece

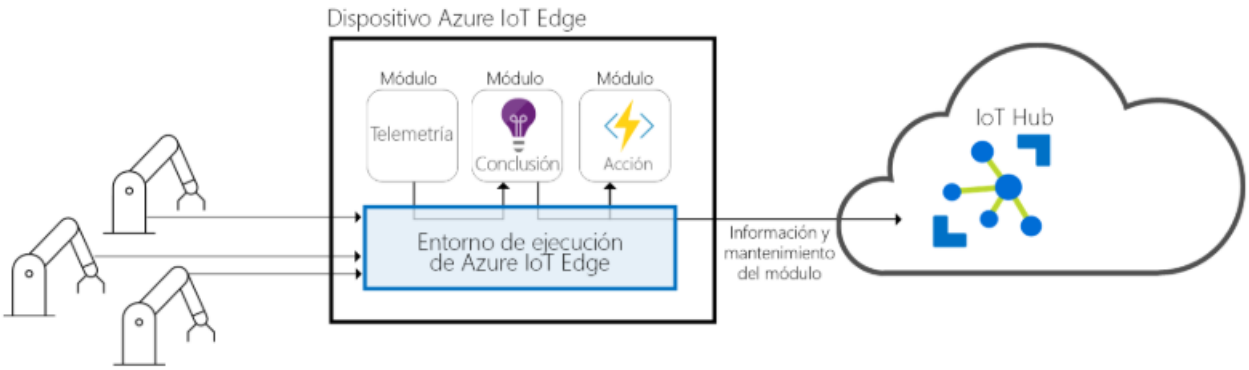


Figura 2.4: Arquitectura de un dispositivo Azure IoT Edge

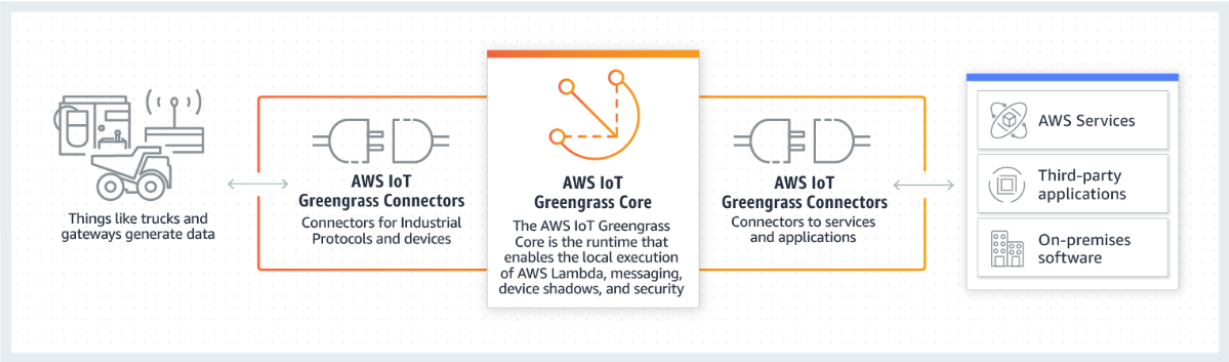


Figura 2.5: Arquitectura de un dispositivo AWS IoT Greengrass

la ventaja de funcionar sin conexión y de hacer uso de comunicaciones seguras, gracias al cifrado tanto en las comunicaciones locales como en la nube.

Un dispositivo AWS *IoT Greengrass* funciona como un centro que puede comunicarse con otros dispositivos que estén debidamente configurados. Estos son aquellos que ejecuten Amazon FreeRTOS, el sistema operativo para microcontroladores propio de AWS, o que cuenten con el SDK de AWS IoT.

Dentro del propio dispositivo AWS *IoT Greengrass* se ejecutan los servicios de AWS necesarios para realizar tareas de administración, aprovisionamiento, predicciones basadas en modelos de aprendizaje automático u otras tareas pertinentes. Puede además conectarse con aplicaciones de terceros.

2.2. Dispositivos de borde

El dispositivo de borde es aquel que concentra toda la funcionalidad del sistema: ingesta, almacenamiento y *forwarding* de datos. En el contexto de un despliegue *Edge Computing*, deberá además realizar todas las tareas de análisis.

A continuación, van a presentarse distintos *gateways* que reúnen las condiciones necesarias para que puedan cumplir la función de dispositivo de borde. Estas condiciones pasan por:

- Bajo consumo energético: El dispositivo estará operativo las 24 horas del día, por lo que resulta vital que no dispare el consumo energético.
- Mínima capacidad de cómputo: El dispositivo debe ser capaz de ejecutar tareas de análisis de datos, mediante el uso de librerías como *Tensorflow* [27] o similares.

La tabla 2.1 muestra una comparativa entre distintos candidatos a ser el dispositivo de borde.

Todos los candidatos presentados equipan arquitectura ARM *multi-core* y tienen una cantidad de memoria RAM aceptable, por lo que cumplen con el requisito de mínima capacidad de cómputo.

Además, guardan en común la característica de tratarse de ordenadores de placa única (*Single Board Computer*, SBC). Estos dispositivos tienen una capacidad computacional más limitada que un ordenador personal, pero suficiente para implementar las tareas objetivo de un dispositivo de borde. Por el contrario, esta limitación provoca que su consumo energético sea significativamente menor al de estos ordenadores, por lo que los convierten en perfectos candidatos a ser el dispositivo de borde seleccionado.

La decisión final pasó por escoger la Banana Pi R1 -observable en la figura 2.6- porque tiene múltiples interfaces de red, lo cual beneficiaría a las tareas de *networking*.

	Banana Pi R1	**Banana Pi R2**	**Banana Pi M64**	**Raspberry Pi 2 Model B**
CPU	A20 ARM Cortex-A7 Dual-Core	MediaTek MT7623N, Quad-Core ARM Cortex-A7	Allwinner 64 Bit Quad Core ARM Cortex-A53 1.2 GHz	900MHz Quad-Core ARM Cortex-A7
GPU	Mali 400 MP2	Mali 450 MP4	Dual-Core Mali 400 MP2	Broadcom VideoCore IV
Memoria	1GB DDR3	2GB DDR3 SDRAM	2GB DDR	1GB RAM
Almacenamiento	MicroSD, SATA Interface	8GB eMMC flash onboard, MicroSD, 2xPuerto SATA 2.0	8GB eMMC flash onboard, MicroSD	MicroSD
Red	4x10/100 Mbit/s Ethernet, Wi-Fi 802.11 b/g/n	5x10/100/1000 Mbit/s Ethernet (MT7530), Wi-Fi 802.11 b/g/n	10/100/100 Mbit/s Ethernet, Wi-Fi 802.11 b/g/n, Bluetooth 4.0	100 Base Ethernet
Puertos USB	1xUSB 2.0, 1xUSB 2.0 OTG	2xUSB 3.0, 1xUSB 2.0 OTG	2xUSB 2.0, 1xUSB 2.0 OTG	4xUSB 2.0
SO	Android, Linux	OpenWRT, Debian, Ubuntu	Android, Linux	Linux, Windows 10

Cuadro 2.1: Comparación de dispositivos de borde

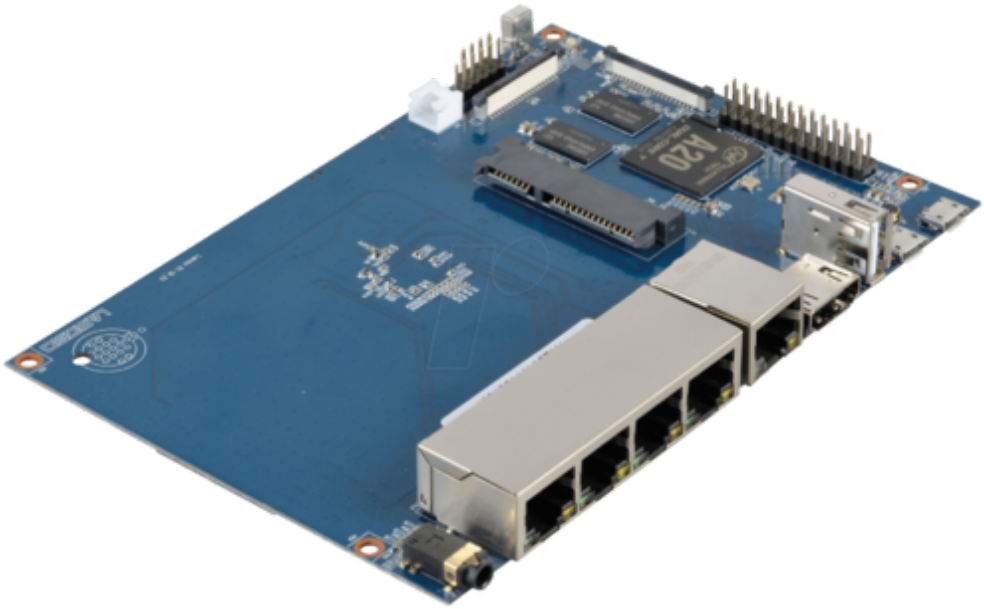


Figura 2.6: Banana Pi R1

2.3. Tecnologías de virtualización y contenerización de servicios

La arquitectura objetivo del proyecto pasa por permitir ejecutar múltiples servicios software dentro del dispositivo, de tal manera que lo hagan de manera independiente. Esta circunstancia permite dar soporte a distintos clientes sin que la privacidad de sus datos se vea comprometida.

Esta encapsulación de servicios puede lograrse por diferentes vías, siendo dos las más utilizadas: los servicios de virtualización ligeros y los hipervisores o máquinas virtuales [19, 18, 16].

Entre estas dos técnicas existen diferencias suficientemente significativas para permitirnos decantarnos por una de ellas: en este proyecto se va a desplegar un servicio de virtualización ligero basado en contenedores, haciendo uso de la plataforma Docker.

Un contenedor es una unidad software aislada, que dispone de su propio subsistema de red, memoria y sistema de ficheros [18]. Así, un contenedor puede ser ejecutado en diferentes sistemas con diferentes sistemas operativos, siendo sencillo construirlo, desplegarlo

y ejecutarlo.

La clave por la cual la virtualización basada en contenedores está ganando enteros es que utiliza el propio *kernel* del sistema para proveerlo a los contenedores; esta manera de gestionar los recursos reduce la sobrecarga producida, sobretodo si se compara con el uso de máquinas virtuales, que virtualizan sus propios recursos hardware y sus *drivers* [21, 20, 29, 19]. La diferencia entre ambas arquitecturas se aprecia en la figura 2.7.

En general, el uso de contenedores conlleva las siguientes ventajas:

- La construcción, instanciación e inicialización de los contenedores es más rápida que el de las máquinas virtuales.
- El tamaño de los contenedores es menor que el de las máquinas virtuales, por lo que dentro de un dispositivo pueden desplegarse una mayor cantidad de contenedores.
- La interconexión y comunicación entre distintos contenedores es fácilmente desplegable.
- Un mal funcionamiento de un contenedor, o cualquier problema derivado del mismo no afecta a la máquina anfitrión, pues se ejecuta de manera independiente y aislada.
- Cualquier procesador es apto para su despliegue, pues no es necesario soporte nativo para su virtualización.

En el contexto de este proyecto, esta última ventaja es clave para poder hacer uso de arquitectuas ARMv7 de 32 bits, que no disponen de soporte nativo para virtualización, por lo que el uso de contenedores es necesario.

2.3.1. Docker como servicio de virtualización ligero

Docker es una plataforma para desarrollar, exportar y ejecutar aplicaciones. Permite aislar las aplicaciones para agilizar todo el proceso de desarrollo y publicación, y también facilita la ejecución de múltiples contenedores en una misma distribución hardware.

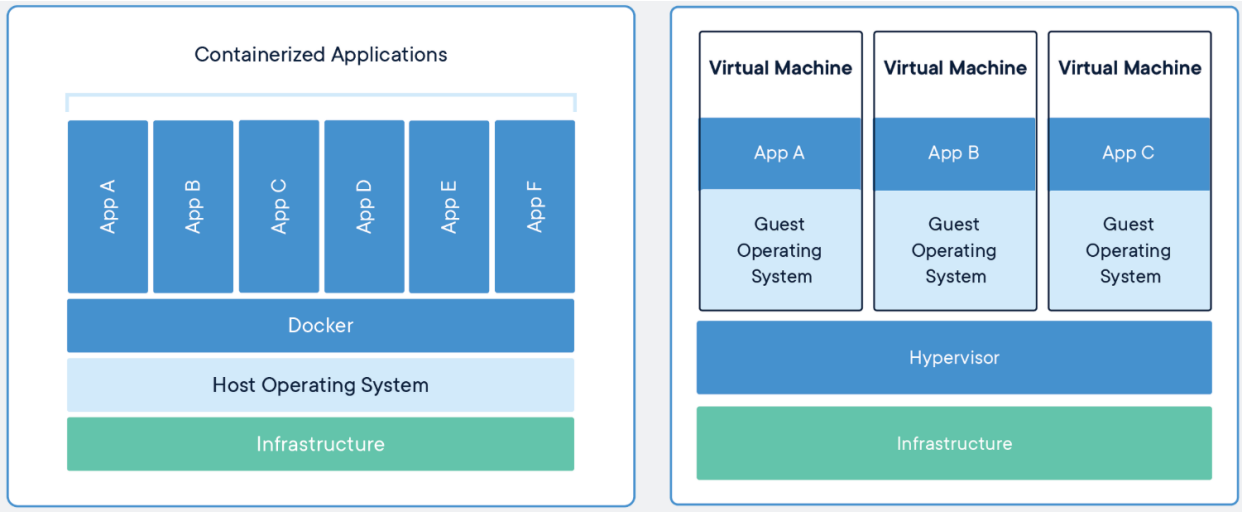


Figura 2.7: Comparativa entre contenedores y máquinas virtuales

Un contenedor encapsula el código y todas sus dependencias de tal manera que una aplicación puede ejecutarse de manera rápida y segura de un entorno de programación a otro. Es un paquete ejecutable ligero y único que incluye todo lo que se necesita para que la aplicación se pueda ejecutar: código, tiempo de ejecución, herramientas del sistema, bibliotecas y configuración [29, 9].

Docker basa su funcionamiento en *Docker engine*, que es una aplicación cliente-servidor, observable en la figura 2.8, y compuesta en gran medida por los siguientes tres componentes:

- Un servidor corriendo como demonio, que crea y administra las imágenes, contenedores, redes y volúmenes de datos.
- API REST que conecta con el servicio demonio.
- Interfaz de línea de comandos (*Command line interface, CLI*), a través del comando Docker. Utiliza la API para interaccionar con el demonio, a la cual se conecta a través del cliente.

También provee un repositorio oficial de imágenes, *Docker Hub* [8], de donde pueden obtenerse tanto imágnes oficiales como otras desarrolladas por usuarios.

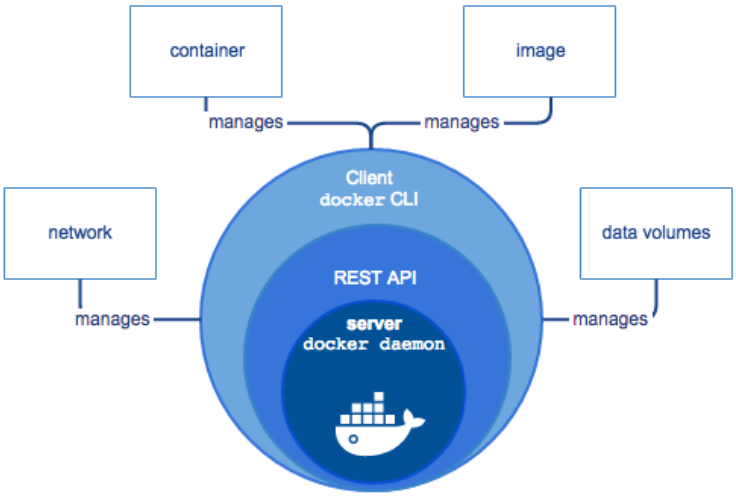


Figura 2.8: Docker engine

Una imagen es una plantilla con permisos de lectura que es utilizada para la creación de contenedores, que son las instancias ejecutables de las imágenes. Determina qué características tendrá la aplicación que la instancie: sistema operativo, servicios, conexiones, etc.

Cuando un contenedor es levantado, se le pueden añadir más parámetros de configuración, pero estos no son persistentes; por lo que una vez el contenedor es eliminado, dichos cambios son eliminados a menos que que hayan sido guardados, gracias a una funcionalidad de la CLI que más adelante será detallada.

La CLI ofrece una gran cantidad de funcionalidades que permiten administrar la infraestructura de contenedores de manera sencilla. A continuación, van a listarse los comandos que más resultarán de utilidad para la gestión del sistema de este proyecto:

Docker run

El comando tiene la siguiente estructura:

`docker run <opciones><imagen><comando>`

El comando que levanta contenedores, construidos en base a la imagen indicada en el comando. Primeramente comprueba que la imagen está almacenada localmente; en caso de

no estarlo, la busca en *Docker Hub* y la descarga.

Cuando ejecuta el contenedor, lo conecta a la red por defecto, que es la conexión de red que tiene la máquina anfitrión, y le asigna una IP. Esta configuración, no obstante, es modificable en base a una serie de parámetros.

Ofrece una serie de parámetros para administrar de manera más detallada la configuración del contenedor. Permite, por ejemplo, mapear los puertos de la máquina anfitrión con los propios del contenedor, de manera que se puede realizar una gestión de la red más completa. También puede decidirse si, una vez finaliza su ejecución, se elimina el contenedor o se deja almacenado para poder ejecutarlo de nuevo más adelante. Otra configuración útil es elegir cuántas *cores* de la CPU de la máquina anfitrión se le dedican al contenedor, siendo esta configuración modificable dinámicamente.

Docker start/stop

El comando tiene la siguiente estructura:

```
docker start/stop <contenedor><opciones>
```

Con el comando *stop* se puede detener la ejecución de un contenedor que esté en funcionamiento. El comando *start* reaunada la ejecución.

Docker exec

El comando tiene la siguiente estructura:

```
docker exec <opciones><contenedor><comando>
```

Ejecuta un comando en el contenedor que está ejecutándose.

Docker cp

El comando tiene la siguiente estructura:

```
docker cp <opciones><contenedor>:<ruta origen><ruta destino>
```

```
docker cp <opciones><ruta origen><contenedor>:<ruta destino>
```

Copia un archivo o bien desde la máquina anfitrión a un contenedor, o viceversa.

Docker commit

El comando tiene la siguiente estructura:

`docker commit <opciones><contenedor><imagen>`

Genera una nueva imagen a partir de un contenedor. Mediante este comando es posible hacer persistentes los cambios realizados sobre un contenedor. Al modificar la imagen, el próximo contenedor levantado sobre dicha imagen tendrá la configuración registrada gracias a este comando.

Docker pull/push

El comando tiene la siguiente estructura:

`docker pull/push <opciones><imagen>`

Se conecta con *Docker Hub* para, o bien descargar una imagen con el comando *pull*, o exportarla a dicha plataforma con el comando *push*.

Como se ha señalado con anterioridad, cuando se desea levantar un contenedor a partir de una imagen, lo primero que hace la instrucción *run* es buscar esta imagen localmente. En caso de no encontrarla, ejecuta un comando *pull* para obtenerla del repositorio de imágenes.

Docker rm/rmi

El comando tiene la siguiente estructura:

`docker rm/rmi <opciones><contenedor>/<imagen>`

Elimina un contenedor o una imagen del repositorio local.

Docker stats

El comando tiene la siguiente estructura:

`docker stats <opciones><contenedor>`

Muestra estadísticas en tiempo real del contenedor indicado, o de todos mediante el parámetro -a. Las estadísticas muestran el consumo de CPU de la máquina anfitrión, el uso de memoria, el límite de memoria que puede utilizar, la cantidad total de datos que

ha enviado y recibido a través de la red, la cantidad de datos que ha leído o escrito en bloques de memoria del anfitrión y una lista con los PIDs de los procesos que ha creado el contenedor.

Docker update

El comando tiene la siguiente estructura:

```
docker update <opciones><contenedor>
```

Actualiza la configuración del contenedor indicado. Entre otros parámetros, puede modiciar el número de *cores* dedicados, el total de memoria asignado o el número de procesos que puede crear.

Capítulo 3

ECM. Diseño y funcionalidad

En este capítulo se mostrarán todas las funcionalidades de la solución.

Esta consiste en una plataforma, alojada en el dispositivo de borde, que provee de servicios a sus clientes para que puedan consultar qué datos están recogiendo sus sensores, realizar inferencias sobre estos, o levantar servicios personalizados.

Todos estos servicios son bautizados como servicios ECM (*Edge Computing Manager*). Están disponibles en este repositorio de GitHub [12], bajo una licencia de código abierto *GNU General Public License*.

Todos estos servicios, levantados bajo demanda, se alojarán en contenedores Docker, a los cuales el cliente podrá conectarse vía TCP, a través de una aplicación de consola desarrollada en Python. Las imágenes de estos contenedores están disponibles en este repositorio de Docker Hub [11].

3.1. Registro de cliente

Cuando un cliente se conecta a la plataforma por primera vez, comienza un proceso de registro. Así, el sistema puede identificar unívocamente a dicho cliente, y solo aceptará peticiones de servicios y conexiones a contenedores si se identifica debidamente.

Para registrarse, ha de proveer al sistema de la siguiente información:

- Nombre de cliente: Nombre con el que se identifica el cliente. Es utilizado por la pla-

taforma para identificarlo. Además, en base a este nombre, se genera un *hash* que será proporcionado al cliente para que lo utlice para autenticarse en posteriores conexiones.

El funcionamiento del programa puede apreciarse en la figura 3.1.

```
root@lamobo-r1:~/client_tcp# python3 register_client.py -c myCompany
Received response: b'{"body": "Perfil de cliente registrado con exito. ID de identificacion: \'5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654\'.", "status": "OK"}'
```

Figura 3.1: Registro de cliente

3.2. Levantamiento de servicio

Una vez el cliente se ha registrado con éxito en la plataforma, puede solicitar el levantamiento de un servicio. La plataforma presenta tres servicios distintos:

- Servicio de datos: Este servicio reenvía los datos enviados por los sensores desplegados. Se levanta un contenedor que se pone a escuchar las conexiones de los sensores, para recibir los datos, y la del cliente que quiere recibirlos. El sistema se encarga de reenviar los datos recibidos al cliente conectado, por lo que este obtiene en tiempo real las mediciones de los dispositivos.

- Servicio de análisis: Este servicio levanta una infraestructura similar al del caso anterior. Sin embargo, en vez de reenviar los datos en bruto, realiza inferencias sobre estos y envía al cliente el resultado de la misma. Tiene dos modos de uso: hacer inferencia por cada dato recibido, o por *batches*, de tal manera que el usuario decide el tamaño del mismo.

- Servicio personalizado: El cliente provee una imagen, que debe estar disponible desde Docker Hub.

3.2.1. Servicio de escucha de datos

Como en todo servicio, para hacer uso del mismo el cliente ha de ejecutar dos programas.

El primero consiste en solicitar el levantamiento del contenedor. Para ello, el cliente ha de proveer la siguiente información:

- Clave de autenticación: Se trata del *hash* que generó el servicio de registro en base al nombre de cliente provisto en el proceso de registro.

El funcionamiento de esta primera tarea puede apreciarse en la figura 3.2.

```
root@lamobo-r1:~/client_tcp# python3 cliente.py -s getdata --id 5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654
{"service": {"default_service": "getdata", "custom": 0, "image": "null"}, "cpus": {"max": 1, "min": 0.5}, "id": "5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654", "da
ta": "./data/temp.csv", "batch": 0, "lports": ["default"], "nports": 1, "order": "null", "number_values": 10}
Now waiting for response
Received resp: b'{"body": "Puedes conectarte al contenedor a traves del puerto 5051. El ID del servicio es 9656.", "status": "OK"}'
Tiempo de operacion: 10.01983118057251
```

Figura 3.2: Levantamiento de servicio de lectura de datos

El servidor, entonces, provee al cliente de un puerto a través del cual puede conectarse a su contenedor para recibir los datos.

Lo siguiente que tiene que hacer el cliente es hacer uso de otra aplicación de consola que le permite conectarse al contenedor. En este caso, ha de proveer la siguiente información:

- Clave de autentiación: Para identificarse de manera correcta ante el contenedor.
- Puerto: Debe proveer el puerto que le ha indicado el servidor que ha levantado el contenedor.

El programa, si consigue conectarse con éxito al contenedor, comenzará a mostrar por pantalla los datos que el sensor va enviando al dispositivo. El funcionamiento puede apreciarse en la figura 3.3.

```
root@lamobo-r1:~/client_tcp# python3 container_connection.py --id 5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654 -p 5051
b'{"hum": "56.75093445216663", "temp": "32.216923157010086"}'
b'{"hum": "49.168519078174185", "temp": "31.693343921960636"}{"hum": "58.99818183384083", "temp": "37.02861901339933"}{"hum": "59.716650744818125", "temp": "33
.601386465708416"}{"hum": "67.51234422679241", "temp": "25.74286718909405"}{"hum": "35.50020989050891", "temp": "21.77041279044948"}{"hum": "34.4897472437949",
 "temp": "38.90461129171544"}{"hum": "56.22357665852415", "temp": "29.63221877754009"}{"hum": "36.73022039575227", "temp": "37.51374700583447"}{"hum": "57.9774
64840895294", "temp": "20.672086504254864"}'
b'{"hum": "48.45320191593906", "temp": "31.877024256740846"}'
b'{"hum": "40.62725714636492", "temp": "22.411133365948007"}'
^CInterrupted!
```

Figura 3.3: Lectura de datos desde contenedor

3.2.2. Servicio de análisis de datos

La primera tarea que ha de realizar el cliente es la de solicitar el levantamiento del contenedor. Para ello, ha de proveer la siguiente información:

- Clave de autenticación: Se trata del *hash* que generó el servicio de registro en base al nombre de cliente provisto en el proceso de registro.
- Modelo analítico: Fichero .h5 que contiene el modelo utilizado por el programa que realizará la inferencia sobre los datos.
- Orden de las columnas: El modelo está entrenado sobre un set de datos sin cabecera. Por tanto, el orden de las columnas es importante para que el modelo sepa interpretar los datos que reciba. El cliente debe proveer un pequeño diccionario formateado en JSON que indique este orden. Deberá tener la forma "{key:'value'}", donde la clave será un número entero que funcionará como índice.

Adicionalmente, este servicio funciona en dos modos:

- Modo *batch*: En la modalidad por lotes, el servicio aglutina un número determinado de valores para analizarlos de golpe. En este caso, el cliente ha de proveer dicho número de valores.
- Modo dato a dato: El servicio realiza una inferencia por cada dato que llega de los sensores. En este caso, el cliente no ha de proveer ninguna información extra.

Para el primer caso, el funcionamiento puede apreciarse en la figura 3.4.

```
root@lamobo-r1:~/client_tcp# python3 cliente.py --id 5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654 -s analisis -o "{1:'temp',2:'hum'}" -f model.h5 -k 10 -b 1
{"id": "5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654", "number_values": 10, "lports": ["default"], "service": {"image": "null", "default_service": "analisis", "cus
tom": 0}, "batch": 1, "order": "{1:'temp',2:'hum'}", "data": "./data/temp.csv", "cpus": {"max": 1, "min": 0.5}, "nports": 1}
Now sending file
Now waiting for response
Received resp: b'{"status": "OK", "body": "Puedes conectarte al contenedor a traves del puerto 5051. El ID del servicio es 7230."}'
Tiempo de operacion: 9.984619379043579
```

Figura 3.4: Levantamiento de servicio de análisis de datos en modo batch

Para el segundo caso, el funcionamiento puede apreciarse en la figura 3.5.

```
root@lamobo-r1:~/client_tcp# python3 cliente.py --id 5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654 -s analisis -o "{1:'temp',2:'hum'}" -f model.h5
{"nports": 1, "order": "{1:'temp',2:'hum'}", "number_values": 10, "service": {"custom": 0, "default_service": "analisis", "image": "null"}, "batch": 0, "cpus":
 {"min": 0.5, "max": 1}, "id": "5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654", "lports": ["default"], "data": "./data/temp.csv"}
Now sending file
Now waiting for response
Received resp: b'{"body": "Puedes conectarte al contenedor a traves del puerto 5051. El ID del servicio es 7520.", "status": "OK"}'
Tiempo de operacion: 10.938262462615967
```

Figura 3.5: Levantamiento de servicio de análisis de datos en modo dato a dato

En ambos casos, puede apreciarse el funcionamiento de conectarse al contenedor a través del puerto recibido, en la figura 3.6.

```
root@lamobo-r1:~/client_tcp# python3 container_connection.py --id 5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654 -p 5051
b'{"hum": "58.21739638768911", "temp": "30.50516198957032", "msg": "Alerta por condiciones inapropiadas"}'
b'{"hum": "68.99355640863894", "temp": "22.346573439067228", "msg": "Alerta por condiciones inapropiadas"}'
b'{"hum": "54.50586351028762", "temp": "20.043500735358823", "msg": "Alerta por condiciones inapropiadas"}{"hum": "67.8236763596608", "temp": "31.1766623457055
3", "msg": "Alerta por condiciones inapropiadas"}'
b'{"hum": "65.76989037871101", "temp": "38.22435667600506", "msg": "Alerta por condiciones inapropiadas"}'
b'{"hum": "63.142446379138335", "temp": "25.771513814978242", "msg": "Alerta por condiciones inapropiadas"}'
b'{"hum": "49.82327963382551", "temp": "32.583309385854754", "msg": "Alerta por condiciones inapropiadas"}'
^CInterrupted!
```

Figura 3.6: Lectura de resultados de inferencia de datos

3.2.3. Servicio personalizado

El cliente debe solicitar el levantamiento del contenedor. Para ello, ha de proveer la siguiente información:

- Imagen: Nombre de la imagen a utilizar. Debe de estar disponible en *Docker Hub*.
- Lista de puertos de escucha: Lista de puertos abiertos dentro del contenedor, para que el servidor pueda mapearlos correctamente con los del dispositivo.
- Número de puertos: Número de puertos que hay abiertos. Este valor debe coincidir con la longitud de la lista de puertos.

El funcionamiento de esta tarea puede apreciarse en la figura 3.7.

```
root@lamobo-r1:~/client_tcp# python3 cliente.py --id 5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654 -p 1 -i franpach/fp_analisis -l "5050,5051" -n 2
puertos correctos: 1
{"batch": 0, "lports": ["5050", "5051"], "cpus": {"max": 1, "min": 0.5}, "id": "5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654", "order": "null", "number_values": 10
, "nports": 2, "service": {"image": "franpach/fp_analisis", "default_service": "personalised service", "custom": 1}, "data": "./data/temp.csv"}
Now waiting for response
Received resp: b'{"body": "Puedes conectarte al servicio a traves de los puertos [\'5052\', \'5053\']. El ID del servicio es 4772", "status": "OK"}'
Tiempo de operacion: 9.245407104492188
```

Figura 3.7: Levantamiento del servicio personalizado

3.2.4. Parámetros adicionales

Los tres servicios guardan parámetros comunes. Estos tienen que ver con la capacidad que el cliente quiere que se le dedique al contenedor.

Como se ha visto, Docker permite asignar más o menos *cores* a los contenedores desplegados. Este valor es elegible por el cliente, que más concretamente puede determinar:

- CPU mínima: Mínimo número de *cores* asignable al contenedor.
- CPU máxima: Máximo número *cores* asignable al contenedor. No podrá sobrepasar nunca 1 *core*.

Por defecto, en caso de que el usuario no indique estos valores, se le asigna como mínimo medio *core*, mientras que el valor máximo se pone a 1.

3.3. Consultar servicios activos

El cliente puede consultar qué servicios tiene levantados. Para ello, ha de proveer la siguiente información:

- Clave de autenticación: Se trata del *hash* que generó el servicio de registro en base al nombre de cliente provisto en el proceso de registro.

El funcionamiento de esta tarea puede apreciarse en la figura 3.8.

```
root@lamobo-r1:~/client_tcp# python3 cliente.py -s info --id 5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654
{"order": "null", "batch": 0, "cpus": {"min": 0.5, "max": 1}, "id": "5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654", "service": {"image": "null", "default_service":
 "info", "custom": 0}, "data": "./data/temp.csv", "lports": ["default"], "number_values": 10, "nports": 1}
Now waiting for response
Received resp: b'{"body": "Tienes 1 contenedores ejecutandose: [ID: 9656, CPUs: 1, tipo: listening]", "status": "OK"}'
Tiempo de operacion: 0.03404879570007324
```

Figura 3.8: Consulta de servicios activos

3.4. Detener servicio

El cliente puede solicitar que se detenga un servicio que tenga activo. Para ello, ha de proveer la siguiente información:

- Clave de autenticación: Se trata del *hash* que generó el servicio de registro en base al nombre de cliente provisto en el proceso de registro.

- Identificador del servicio: Identificador unívoco del servicio, que se obtiene cuando se solicita el levantamiento de un contenedor, o mediante la solicitud de información de servicios activos.

El funcionamiento de esta tarea puede apreciarse en la figura 3.9.

```
root@lamobo-r1:~/client_tcp# python3 cliente.py -s stop --id 5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654 -r 3770
{"serviceid": 3770, "service": {"custom": 0, "image": "null", "default_service": "stop"}, "cpus": {"min": 0.5, "max": 1}, "data": "./data/temp.csv", "id": "5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd
9e654", "order": "null", "nports": 1, "number_values": 10, "lports": ["default"], "batch": 0}
Now waiting for response
Received resp: b'{"body": "El servicio 3770 ha sido parado correctamente", "status": "OK"}'
Tiempo de operacion: 4.40792179107666
```

Figura 3.9: Detención de un servicio

3.5. Eliminar servicio

El cliente puede solicitar que se elimine un servicio que tenga activo. Para ello, ha de proveer la siguiente información:

- Clave de autenticación: Se trata del *hash* que generó el servicio de registro en base al nombre de cliente provisto en el proceso de registro.

- Identificador del servicio: Identificador unívoco del servicio, que se obtiene cuando se solicita el levantamiento de un contenedor, o mediante la solicitud de información de servicios activos.

El funcionamiento de esta tarea puede apreciarse en la figura 3.10.

```
root@lamobo-r1:~/client_tcp# python3 cliente.py --id 5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654 -r 4772 -s delete
{"cpus": {"min": 0.5, "max": 1}, "order": "null", "batch": 0, "number_values": 10, "id": "5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654", "serviceid": 4772, "servic
e": {"default_service": "delete", "custom": 0, "image": "null"}, "nports": 1, "lports": ["default"], "data": "./data/temp.csv"}
Now waiting for response
Received resp: b'{"body": "El servicio 4772 ha sido eliminado correctamente", "status": "OK"}'
Tiempo de operacion: 0.7536966800689697
```

Figura 3.10: Eliminación de un servicio

3.6. Eliminar perfil

El cliente puede solicitar que se elimine su perfil de la plataforma. Para ello, ha de proveer la siguiente información:

- Clave de autenticación: Se trata del *hash* que generó el servicio de registro en base al nombre de cliente provisto en el proceso de registro.

El funcionamiento de esta tarea puede apreciarse en la figura 3.11.

```
root@lamobo-r1:~/client_tcp# python3 cliente.py -s unsubscribe --id 5ec43f77e38f1e8c0f8cf6c3b8c91bedfbd9e654
{"serviceid": 0, "batch": 0, "nports": 1, "number_values": 10, "lports": ["default"], "service": {"custom": 0, "default_service": "unsubscribe", "image": "null"}, "id": "5ec43f77e38f1e8c0f8cf
6c3b8c91bedfbd9e654", "data": "./data/temp.csv", "cpus": {"max": 1, "min": 0.5}, "order": "null"}
Now waiting for response
Received resp: b'{"body": "Tu cuenta ha sido existosamente eliminada", "status": "OK"}'
Tiempo de operacion: 4.7694926261901855
```

Figura 3.11: Eliminación de un perfil de cliente

Capítulo 4

ECM. Desarrollo

4.1. Casos de uso

Toda la funcionalidad provista por el sistema va a ser representada y explicada mediante casos de uso. A continuación se van a listar y detallar los casos de uso para el usuario cliente, que se aprecian en la figura 4.1.

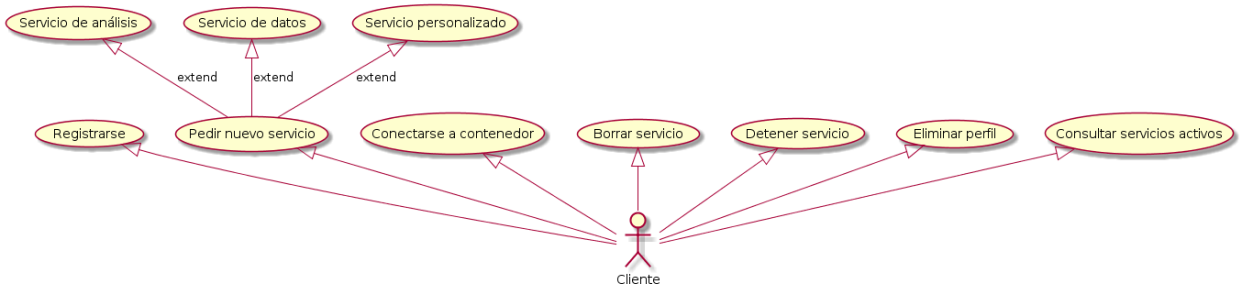


Figura 4.1: Casos de uso del usuario cliente

Registrarse

Cuando un cliente hace uso de la plataforma por primera vez, ha de proveer sus credenciales para ser registrado. En este caso provee un nombre, en base al cual se le genera un identificador único que le es devuelto para que haga uso de él a la hora de autenticarse en toda futura conexión con el sistema.

Solicitar nuevo servicio

El cliente puede solicitar que se levante un nuevo servicio. Estos servicios son:

- Servicio de datos.
- Servicio de análisis.
- Servicio personalizado.

Conectarse a un contenedor

Una vez se levanta un nuevo servicio, el cliente puede conectarse al contenedor levantado en caso de ser necesario. En el caso de los servicios de análisis y recibir datos, la conexión le permitirá recibir el resultado de las tareas analíticas realizadas, o bien los datos recibidos en bruto.

Eliminar servicio

Un cliente puede solicitar que un servicio que tenga levantado deje de ejecutarse, y que el contenedor sea eliminado.

Detener servicio

Un cliente puede solicitar que un servicio que tenga levantado deje de ejecutarse, pero que el contenedor quede almacenado en el sistema para poder volver a ser arrancado con posterioridad.

Consultar servicios activos

Un cliente puede consultar qué servicios tiene levantados, recibiendo un listado con los contenedores que tiene activos y cuántos recursos del sistema tienen asignados cada uno.

Eliminar perfil

Un cliente puede solicitar que su perfil sea eliminado de la plataforma.

4.2. Arquitectura de la solución

A continuación, va a presentare cómo está construida la solución.

Por un lado, van a presentarse todos los componentes globales de la misma, tanto los que componen la parte del sistema que provee el servicio como los que participan externamente.

Por otro lado, va a detallarse cómo está construida la plataforma alojada en el dispositivo de borde.

4.2.1. Diagrama de despliegue del sistema

La solución, que puede apreciarse en la figura 4.2, se construye entorno a los siguientes elementos principales:

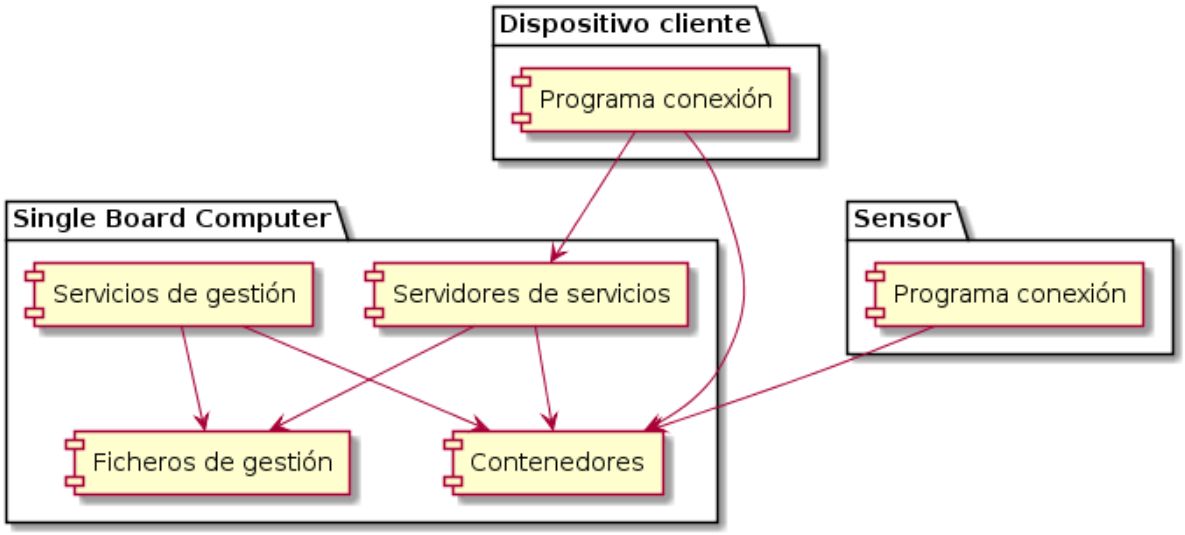


Figura 4.2: Diagrama de despliegue del sistema

- *Single Board Computer*: Se trata del dispositivo de borde. Desplegará los distintos servidores que escuchen las conexiones de los clientes y ejecutará las tareas solicitadas. También contiene servicios internos de gestión para controlar el consumo de recursos del sistema y los perfiles de los clientes.

- Dispositivo del cliente: Se conectará al sistema para solicitar un servicio. Una vez el contenedor sea levantado, se conectará al mismo a través de un puerto que le será comunicado por el servidor que ha levantado el contenedor.

- Sensor: Genera los datos. En su primera conexión, un dispositivo físico se conecta al servidor de *bootstrap* para que se le comunique a qué puerto enviar los datos. Cuando lo recibe, comienza a enviar datos a dicha dirección.

4.2.2. Diseño del servicio

Dentro del dispositivo de borde se aloja la solución completa que provee el sistema de gestión de contenedores y levantamiento de servicios.

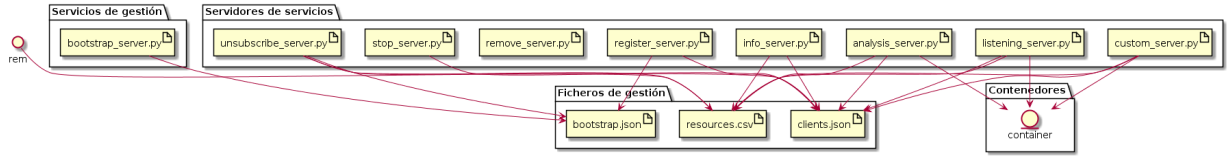


Figura 4.3: Diseño del sistema en el dispositivo de borde

La solución, apreciable en la figura 4.3, la componen cuatro principales grupos de componentes:

- Servidores de servicios: Se trata de servidores, desarrollados en Python, que están constantemente a la escucha de conexiones por parte de clientes que soliciten diferentes servicios. Cuando recibe una conexión, gestiona la solicitud del cliente: bien puede ser aceptada y el servicio ser levantado; o bien puede existir algún problema, en cuyo caso se informará al cliente de qué ha sucedido. En ambos casos, quedando la respuesta enviada, vuelve a ponerse a la escucha de nuevas conexiones.

- Servicios de gestión: Son programas, desarrollados en Python, que gestionan internamente el sistema.

- Ficheros de gestión: Almacenan toda la información del sistema: qué clientes hay registrados, qué servicios tiene activos cada cliente e información de utilidad para el servidor *bootstrap*.
- Contenedores: Son los servicios levantados bajo demanda.

Cada uno de los servicios está descrito a continuación.

Listening_server.py

Escucha conexiones de clientes que deseen levantar un contenedor de envío de datos. Hace uso del fichero *resources.csv* para comprobar que hay recursos disponibles y registrar el servicio, y del fichero *clients.json* para comprobar que la conexión proviene de un cliente registrado, y qué puertos tiene asociados.

Analysis_server.py

Escucha conexiones de clientes que deseen levantar un contenedor de análisis de datos. Hace uso del fichero *resources.csv* para comprobar que hay recursos disponibles y registrar el servicio, y del fichero *clients.json* para comprobar que la conexión proviene de un cliente registrado, y qué puertos tiene asociados.

Custom_server.py

Escucha conexiones de clientes que deseen levantar un contenedor personalizado. Hace uso del fichero *resources.csv* para comprobar que hay recursos disponibles y registrar el servicio, y del fichero *clients.json* para comprobar que la conexión proviene de un cliente registrado, y qué puertos tiene asociados.

Register_server.py

Escucha conexiones de clientes que deseen registrarse en la plataforma. Hace uso de los ficheros *clients.json* y *bootstrap.json* para, primeramente, comprobar que el nombre provisto está disponible y, posteriormente, registrar su información.

Info_server.py

Escucha conexiones de clientes que deseen consultar qué contenedores tienen levantados. Hace uso del fichero *resources.csv* para comprobar qué contenedores tiene levantados el cliente, y del fichero *clients.json* para comprobar que la conexión proviene de un cliente registrado.

Remove_server.py

Escucha conexiones de clientes que deseen eliminar un contenedor que tengan activo. Hace uso del fichero *clients.json* para comprobar que la conexión proviene de un cliente registrado, y del fichero *resources.csv* para comprobar que el servicio existe y, en ese caso, eliminarlo.

Stop_server.py

Escucha conexiones de clientes que deseen detener la ejecución de un contenedor que tengan activo. Hace uso del fichero *clients.json* para comprobar que la conexión proviene de un cliente registrado.

Unsubscribe_server.py

Escucha conexiones de clientes que deseen eliminar su perfil de la plataforma. Hace uso de los tres ficheros de gestión para eliminar todos los servicios, puertos e información asociados a su perfil.

Bootstrap_server.py

Escucha conexiones de sensores que envían datos, para indicarles a qué puerto enviar la información. Hace uso del fichero *bootstrap.json* para consultar el puerto o puertos asociados al sensor que ha realizado la conexión.

Resources.csv

Registra cada servicio que es levantado, junto al cliente que lo ha solicitado, los *cores* que tiene asignados, y en qué límites ha indicado el cliente que puede oscilar dicho valor. Es consultada por los servidores para que comprueben si pueden dar cabida a nuevos servicios, y por el fichero *remove_stopped.py* para hacer limpia de servicios cuya ejecución ha finalizado.

Clients.json

Guarda información de cada cliente registrado: su nombre y qué puertos tiene asignados. Es consultado por los servidores para comprobar que las conexiones están correctamente autenticadas, y si los puertos asignados al cliente están disponibles para proporcionarles conexión al contenedor. También es utilizado por el servidor de registro para guardar la información del nuevo cliente.

Bootstrap.json

Guarda información de los puertos asociados a cada cliente para que el servidor de *bootstrap* provea dicha información a los sensores que se conectan al sistema.

4.3. Diagramas de secuencia

A continuación van a mostrarse los diagramas de secuencia que ilustran los casos de uso anteriormente citados.

Mecánica general

El cliente solicita un nuevo servicio al servidor corrrespondiente. En caso de que las condiciones sean adecuadas -que el cliente se haya registrado correctamente y haya *cores* y puertos disponibles-, se levanta el nuevo contenedor.

El propio servidor envía al usuario los parámetros de conexión, que será el puerto o puertos del anfitrión mapeados a los puertos abiertos del contenedor.

A partir de ahí, el cliente puede conectarse al contenedor vía TCP.

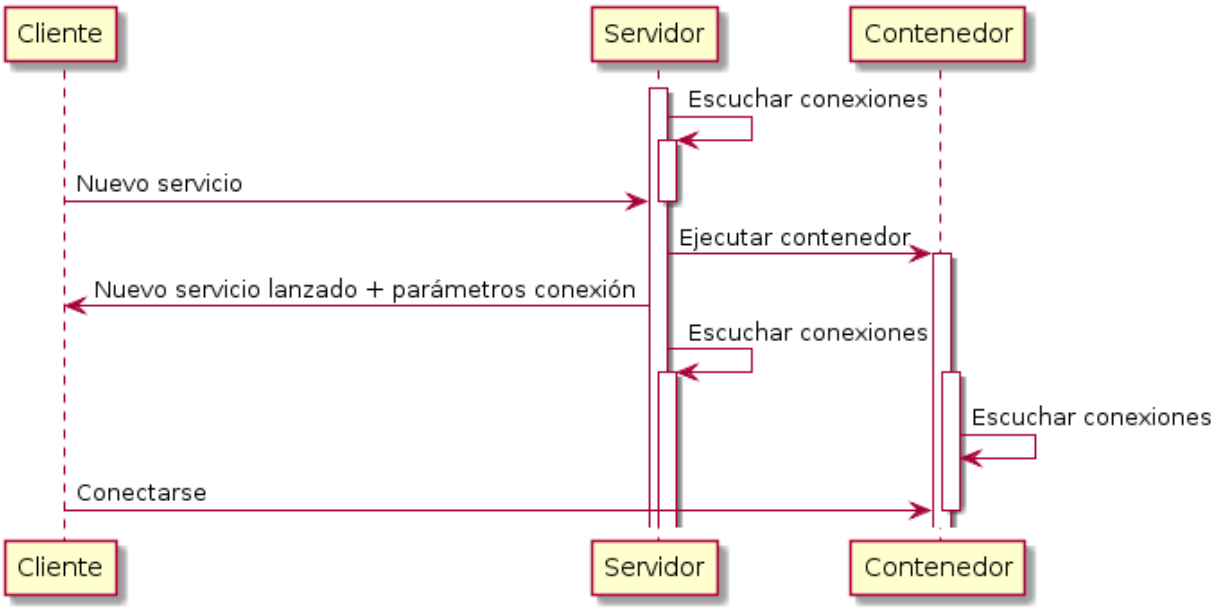


Figura 4.4: Diagrama de secuencia del funcionamiento general del sistema

El diagrama puede apreciarse en la figura 4.4.

Registrar cliente

El cliente solicita registrarse en la plataforma, proveyendo un nombre.

El servidor, primeramente, consulta la lista de clientes para comprobar que no existe otro cliente con el mismo nombre. En caso de no existir, genera una clave hash en base al nombre provisto; dicha clave será la que tendrá que incluir en toda conexión posterior al registro, tanto a la hora de solicitar servicios como los propios sensores a la hora de enviar datos.

Posteriormente, le asigna una serie de puertos dedicados exclusivamente a todo servicio solicitado por él. Lo registra en el sistema y le notifica que el proceso ha finalizado con éxito, y cuál es su cadena de conexión.

El diagrama puede apreciarse en la figura 4.5.

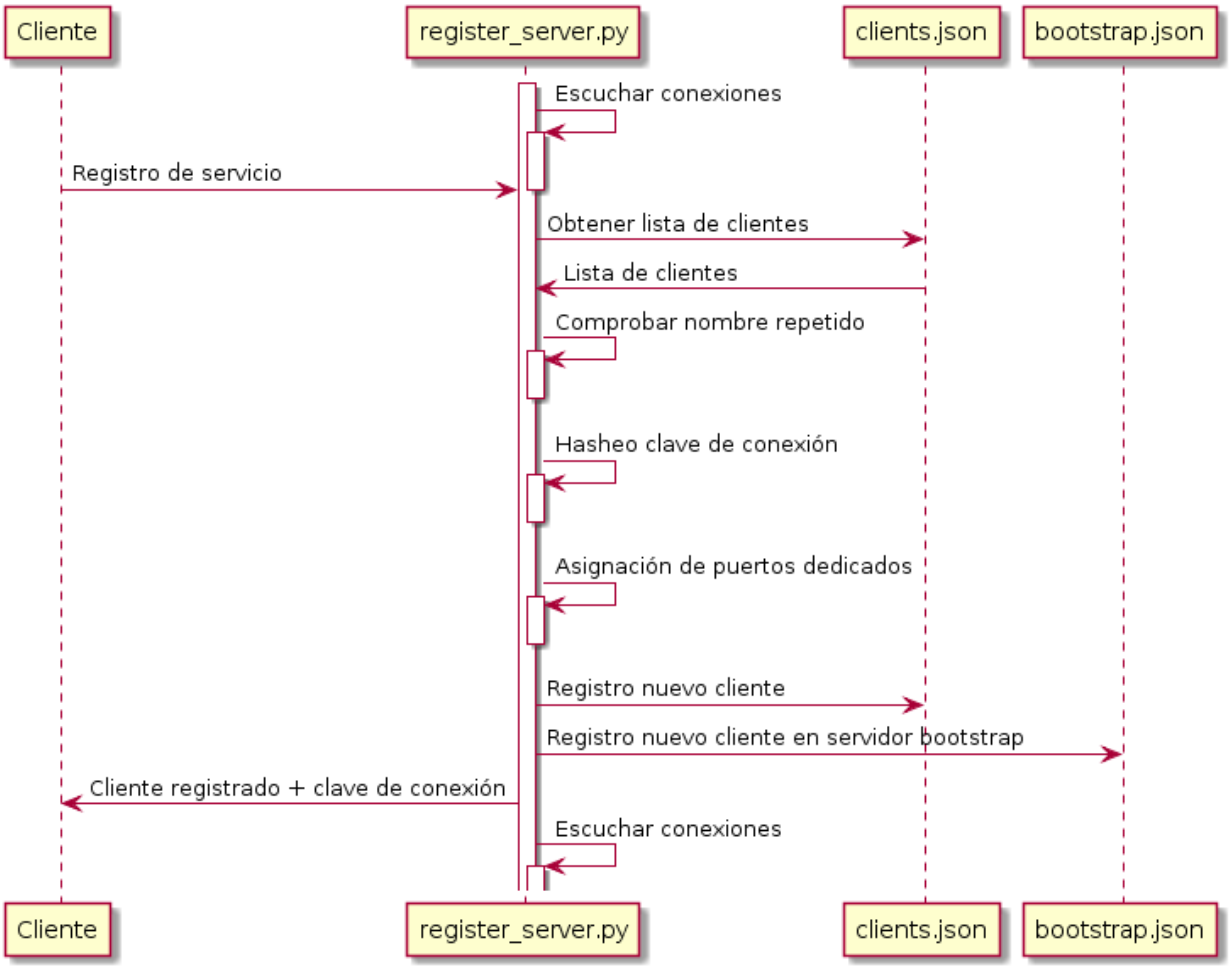


Figura 4.5: Diagrama de secuencia del registro de cliente

Solicitar servicio de datos

El cliente solicita el levantamiento de un contenedor que ejecute el servicio de escucha de datos.

El servidor comprueba que la conexión proviene de un cliente registrado en la plataforma, consultando la clave de autenticación provista. Después comprueba que hay recursos suficientes para dar cabida al nuevo contenedor. En caso de ser posible, prosigue con el lanzamiento del servicio, esta vez comprobando que los puertos dedicados al cliente están disponibles.

Si todo ha funcionado, genera un alias para el servicio, para registrarlo en el registro de

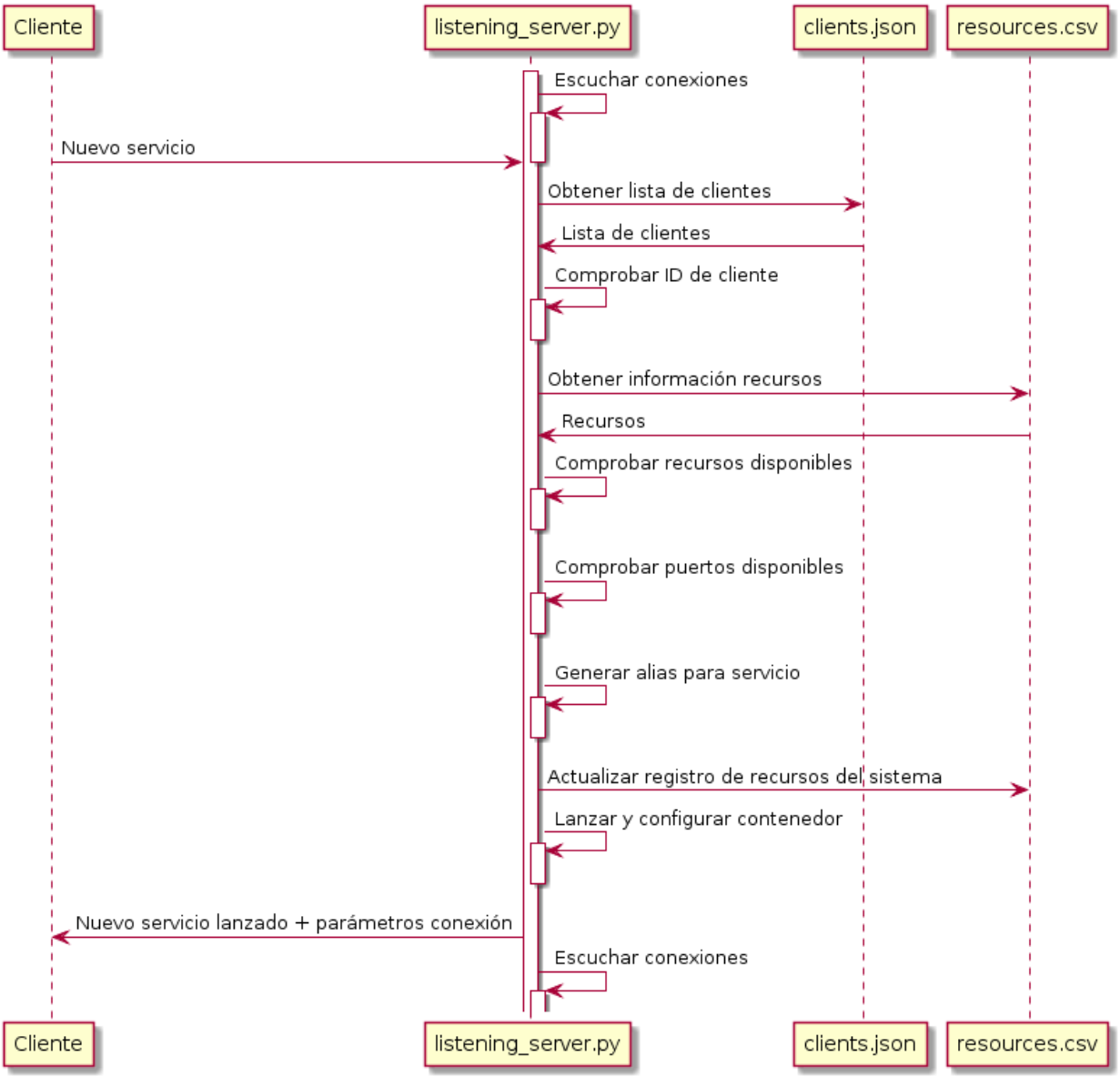


Figura 4.6: Diagrama de secuencia del servicio de datos

recursos del sistema. Informará al cliente de que el servicio ha sido lanzado, y a través de qué puertos puede conectarse.

El diagrama puede apreciarse en la figura 4.6.

Solicitar servicio de análisis

Este servicio funciona de igual manera que el servicio de datos, proveyendo el cliente su clave de autenticación y comprobando el servidor que es posible desplegar el servicio.

En caso de poder levantar el contenedor, el servidor solicita al cliente que le provea el modelo analítico. El cliente lo envía y el servidor crea el contenedor, levantando el servidor que hará uso de dicho modelo para realizar inferencias sobre los datos que vaya recibiendo.

De nuevo, actualizará los registros de recursos e informará al cliente de que la operación ha resultado un éxito, y a través de qué puerto puede conectarse.

El diagrama puede apreciarse en la figura 4.7.

Solicitar servicio personalizado

De nuevo, el servidor comprueba la viabilidad para levantar un contenedor. En caso de poder hacerlo, se descarga la imagen provista por el cliente de *Docker Hub*, gracias al comando *Docker pull*.

En caso de existir dicha imagen, levanta el contenedor y provee al cliente de un mensaje informando de que el servicio está disponible, y a través de qué puertos puede conectarse.

El diagrama puede apreciarse en la figura 4.8.

Consultar servicios activos

El cliente solicita información de los contenedores que tiene levantados. El servidor consulta entonces el fichero *resources.csv*, que mantiene el registro de todos los contenedores activos.

En caso de que tenga servicios registrados a su nombre, se le indica, para cada uno, qué *cores* tiene asignados y su número de identificación.

El diagrama puede apreciarse en la figura 4.9.

Solicitar eliminación de servicio

El cliente provee al servidor el identificador de contenedor que quiere eliminar. El servidor comprueba que existe en el registro de servicios activos, lo elimina y reparte los *cores* liberados entre el resto de contenedores.

El diagrama puede apreciarse en la figura 4.10.

Solicitar detención de servicio

El cliente provee al servidor el identificador de contenedor que quiere detener. El servidor comprueba que se está ejecutando, y lo detiene.

El diagrama puede apreciarse en la figura 4.11

Solicitar eliminación de perfil

El cliente provee al servidor de su clave de autenticación para solicitar que se elimine su perfil del sistema.

El servidor comprueba que la clave es correcta y, en dicho caso, procede a eliminar todos los contenedores que tiene activos y toda información del cliente alojada en los ficheros de gestión.

El diagrama puede apreciarse en la figura 4.12.

Servidor *bootstrap*

Cuando un dispositivo inicia su ejecución, desconoce a qué puerto enviar los datos que sus sensores toman. Por tanto, envían un primer mensaje al servidor *bootstrap*, que comprueba que la clave de autenticación está registrada y, en ese caso, le devuelve el puerto al que enviar los datos.

Posteriormente, el dispositivo envía sus datos a dicho puerto.

El diagrama puede apreciarse en la figura 4.13.

4.4. Gestión de recursos

Resulta conveniente detallar cómo se ha desarrollado la gestión de los recursos del sistema.

Docker no provee mecanismos para controlar la asignación de recursos por encima de las capacidades de la máquina anfitrión. Por tanto, si no se hubiese desarrollado este sistema, los servidores hubieran levantado contenedores de tal manera que, de recibir muchas solicitudes y aceptar todas, la máquina no podría soportar tal carga de trabajo.

De hecho, como se verá en el siguiente capítulo, la ejecución de la máquina se detiene una vez se sobrepasa el número máximo de *cores* asignables. Es por ello que se ha desarrollado un sistema que controla la asignación de recursos.

El sistema está desarrollado para que los recursos asignados a cada contenedor fluctúen según la demanda que exista. Al determinarse unos *cores* mínimos y máximos para cada contenedor, el valor asignado a cada uno podrá reducirse hasta el valor mínimo en caso de que haya mucha actividad en la máquina, y podrá crecer de nuevo hasta el valor máximo si se van dando servicios de baja.

Este sistema se divide en dos partes, según llegue una nueva solicitud de servicio o se de baja otro.

4.4.1. Solicitud de servicio

Cada código de servidor contiene un algoritmo cuyo funcionamiento se ha representado en la figura 4.14.

Cuando se solicita un nuevo servicio, se comprueba que se puede dar cabida al servicio. Se considera que un servicio puede darse de alta en caso de que pueda ejecutarse, al menos, con sus mínimos *cores* asignables.

El algoritmo comprueba cuántos *cores* disponibles hay. Se tiene en cuenta que los *cores* disponibles son aquellos que están sin asignar sumados a un sumatorio de todas los *cores* que se pueden quitar a los servicios desplegados, hasta que todos ellos vean reducida su

asignación al valor mínimo marcado para cada uno.

En caso de haber espacio, el servicio queda aceptado.

El siguiente paso es comprobar si se le puede asignar su máximo valor de *cores* sin reducir los *cores* asignados al resto de contenedores. En ese caso, se le asigna el valor máximo y se levanta.

En caso de no poderse, se le asigna un valor de *cores* intermedio, calculado con la media de los valores mínimo y máximo. En caso de haber *cores* libres disponibles para ese valor, se levanta el servicio.

En caso contrario, se le asigna el valor mínimo. Si sigue sin haber *cores* libres disponibles, lo que se hace es calcular cuántos *cores* adicionales necesita. Ese valor se divide entre el número de servicios activos y, a cada uno de ellos, se le resta ese valor.

En caso de que a alguno de estos servicios no se le pueda restar la totalidad del valor calculado, se le reducirán los *cores* hasta los mínimos asignables. Para cada servicio en esta situación, se sumará el valor que no ha podido restarse. Entonces, esta suma se dividirá entre el número de servicios que todavía no tienen el valor mínimo asignado, y se repetirá este proceso hasta que los *cores* requeridos por el nuevo servicio queden liberados.

Cuando este proceso haya terminado, los *cores* habrán quedado repartidos entre todos los servicios levantados, incluido el último solicitado.

4.4.2. Eliminación de servicio

Como se ha visto, cuando un servicio es solicitado, si resulta necesario a los contenedores ya levantados se les quita *cores* asignados.

De igual manera, cuando un servicio finaliza su ejecución -cuando el cliente así lo solicita- los *cores* que quedan libres son repartidos entre los servicios levantados.

En este caso, el servidor que escucha solicitudes de eliminado contiene un algoritmo representado en la figura 4.15.

El algoritmo comprueba cuántos *cores* han quedado libres, consultando el registro co-

rrespondiente en el fichero *resources.csv* y, seguidamente, divide el valor entre el número de servicios levantados. A cada uno le asigna la parte correspondiente.

En caso de que la suma de los *cores* asignados más los que ha dejado libres el contenedor eliminado superase el valor máximo asignable para un servicio concreto, a este se le asignaría dicho valor.

Los *cores* que no han podido ser asignados debido a esta restricción no se vuelve a dividir entre los servicios que todavía pueden recibir más recursos, para no sobreexplotar la máquina.

4.5. Modelado y construcción de servicios

En esta sección se va a describir cómo se han construido los dos servicios por defecto que ofrece la plataforma: el de redirección de datos y el de inferencia.

Por un lado, se describirá la construcción del contenedor, señalando qué imagen base se ha utilizado para cada uno, y qué servicios se han agregado para hacer funcionarlos; y por otro lado se detallará la lógica que sostiene ambos servicios.

4.5.1. Construcción de los contenedores

En ambos casos, se ha partido de una imagen ya creada, por lo que no se ha hecho uso del servicio de Docker que permite la generación de imágenes desde 0 utilizando un fichero.

Servicio de escucha de datos

Este servicio se ha construido partiendo de la imagen oficial de Ubuntu. Sobre esta imagen base se ha instalado Python3 para la ejecución del servidor.

Servicio de análisis de datos

La arquitectura de la Banana Pi es de 32 bits, por lo que el uso de la librería *Tensorflow* parece, *a priori*, incompatible, atendiendo a la documentación de la misma.

Sin embargo, existe una imagen que, partiendo de una imagen Ubuntu, tiene instalada la librería, pudiendo ser utilizado en dispositivos con arquitecturas ARM de 32 bits.

Dicha imagen se puede encontrar aquí [15].

4.5.2. Diseño de los servicios

Ambos comparten una arquitectura común, apreciable en la figura 4.16.

En el contenedor se despliega un servidor, escrito en Python, que inicia dos hilos. Cada hilo despliega un servidor: uno escucha conexiones por parte de los sensores que envían datos, mientras que el otro escucha la conexión del cliente.

Además, se crea una cola. A esta cola, el servidor de datos reenviará la información pertinente, mientras que el servidor de clientes se pondrá a consumir los datos que vayan llegando a esta y reenviará los datos al cliente, una vez se haya conectado.

Cuando los datos comienzan a llegar al servidor, dependiendo del servicio, el servidor de datos realiza una tarea:

- Servicio de reenvío de datos: El servidor reenvía el dato en bruto a la cola.
- Servicio de análisis: El servidor realiza la inferencia -en caso de ser en modo *batch*, acumula tantos dato como se le haya indicado y los analiza; en caso de ser en modo dato a dato, realiza una inferencia cada vez que llega un dato-, y envía el resultado a la cola.

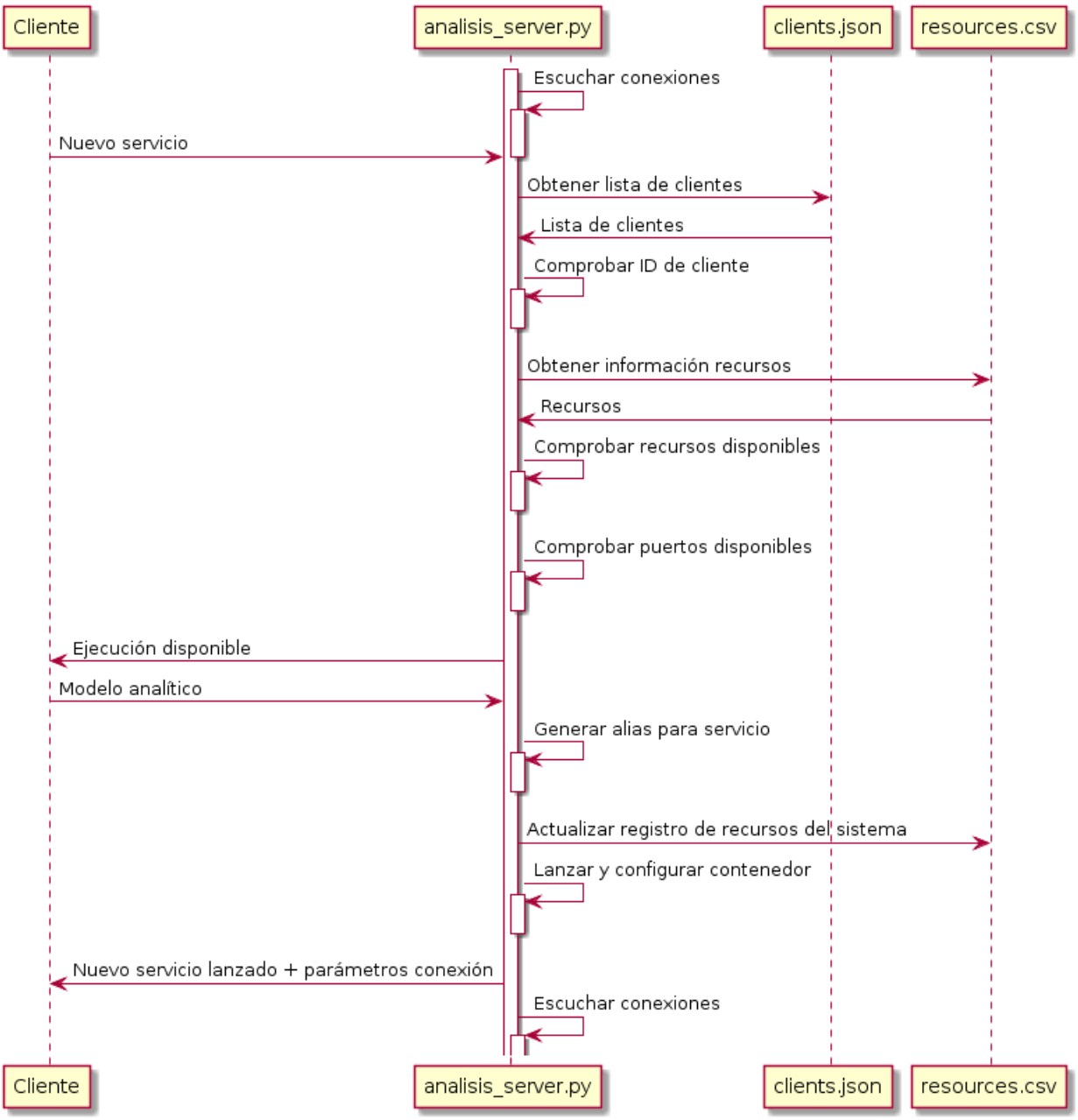


Figura 4.7: Diagrama de secuencia del servicio de análisis

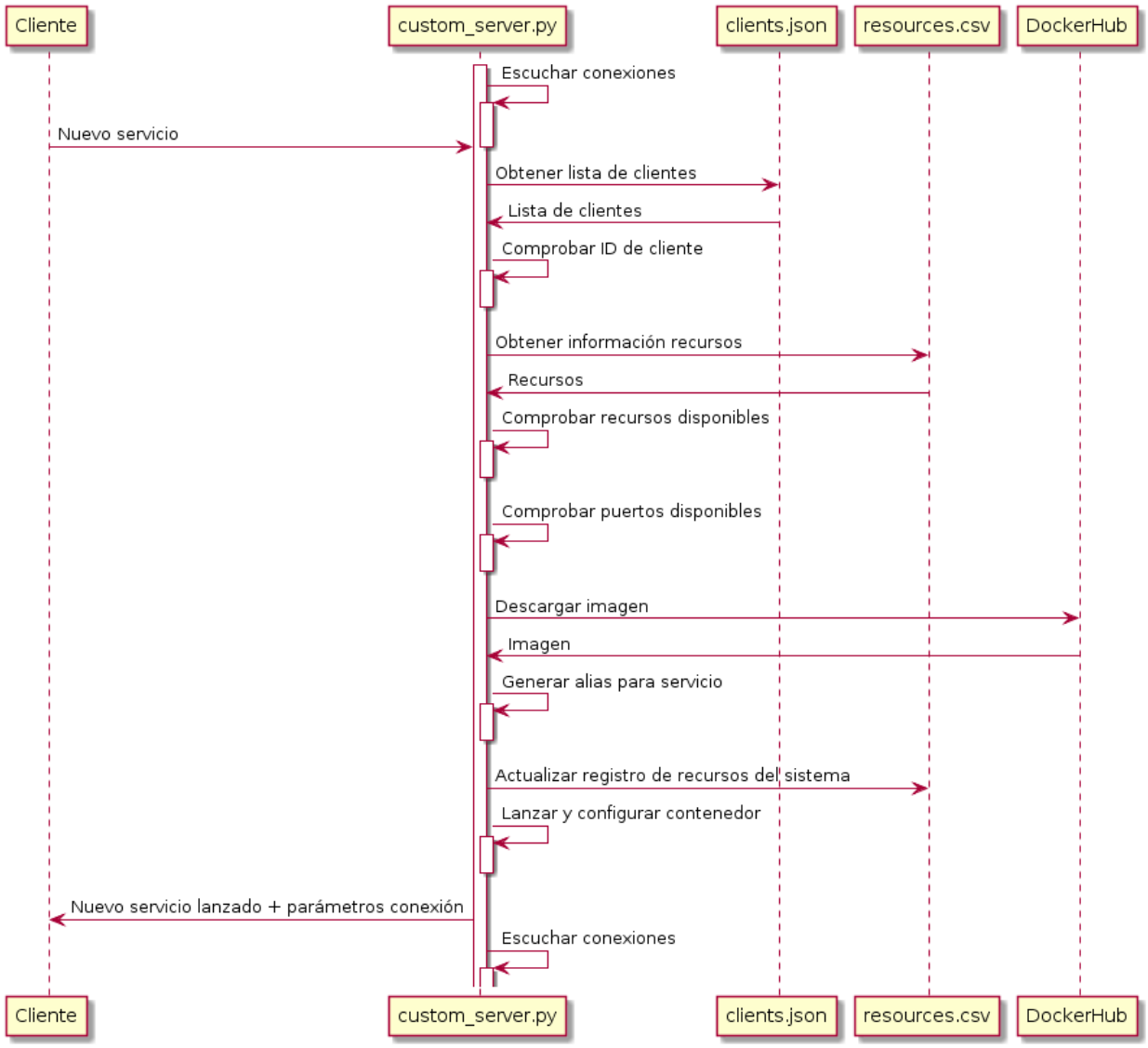


Figura 4.8: Diagrama de secuencia del servicio personalizado

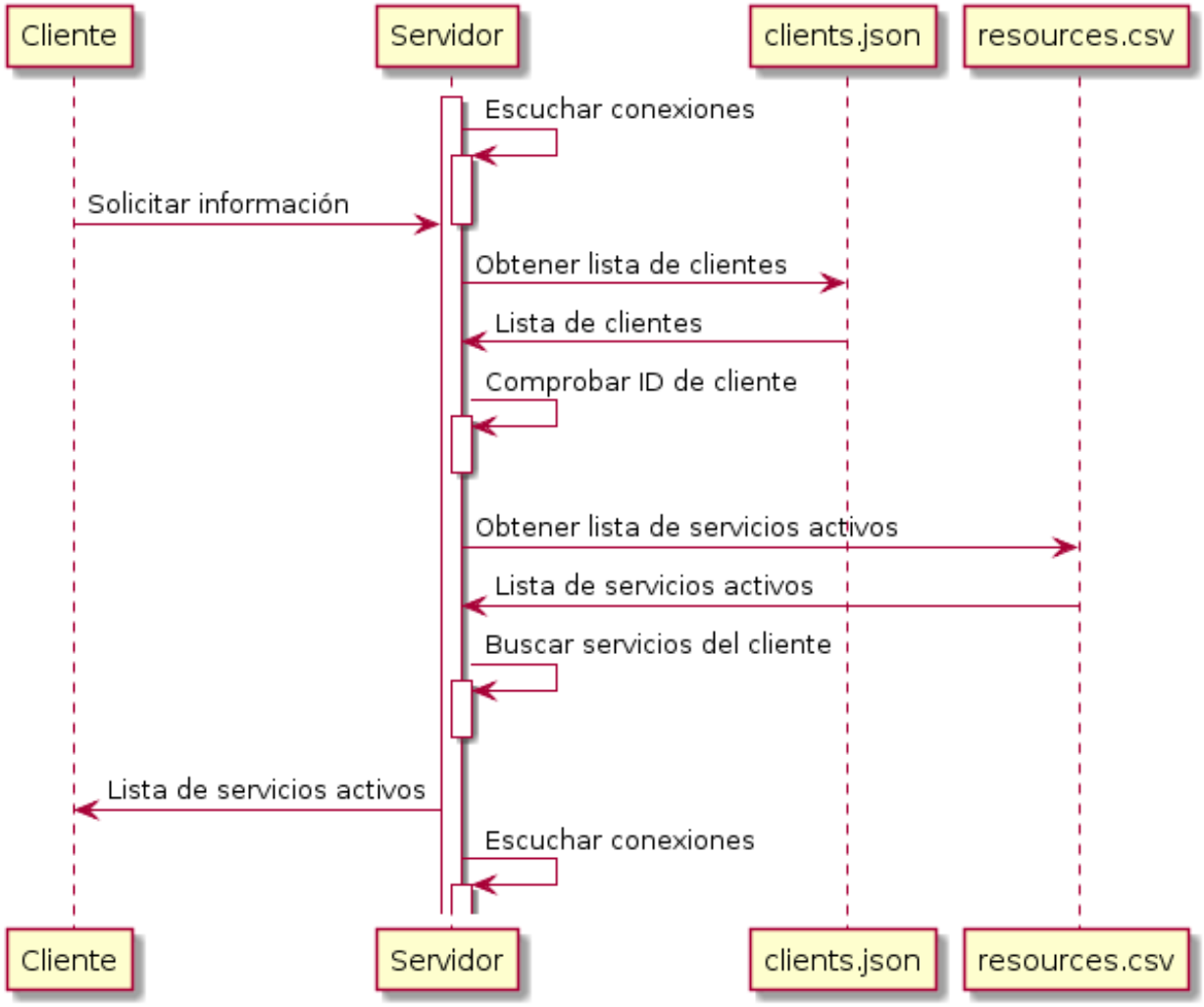


Figura 4.9: Diagrama de secuencia del servicio de información de servicios activos

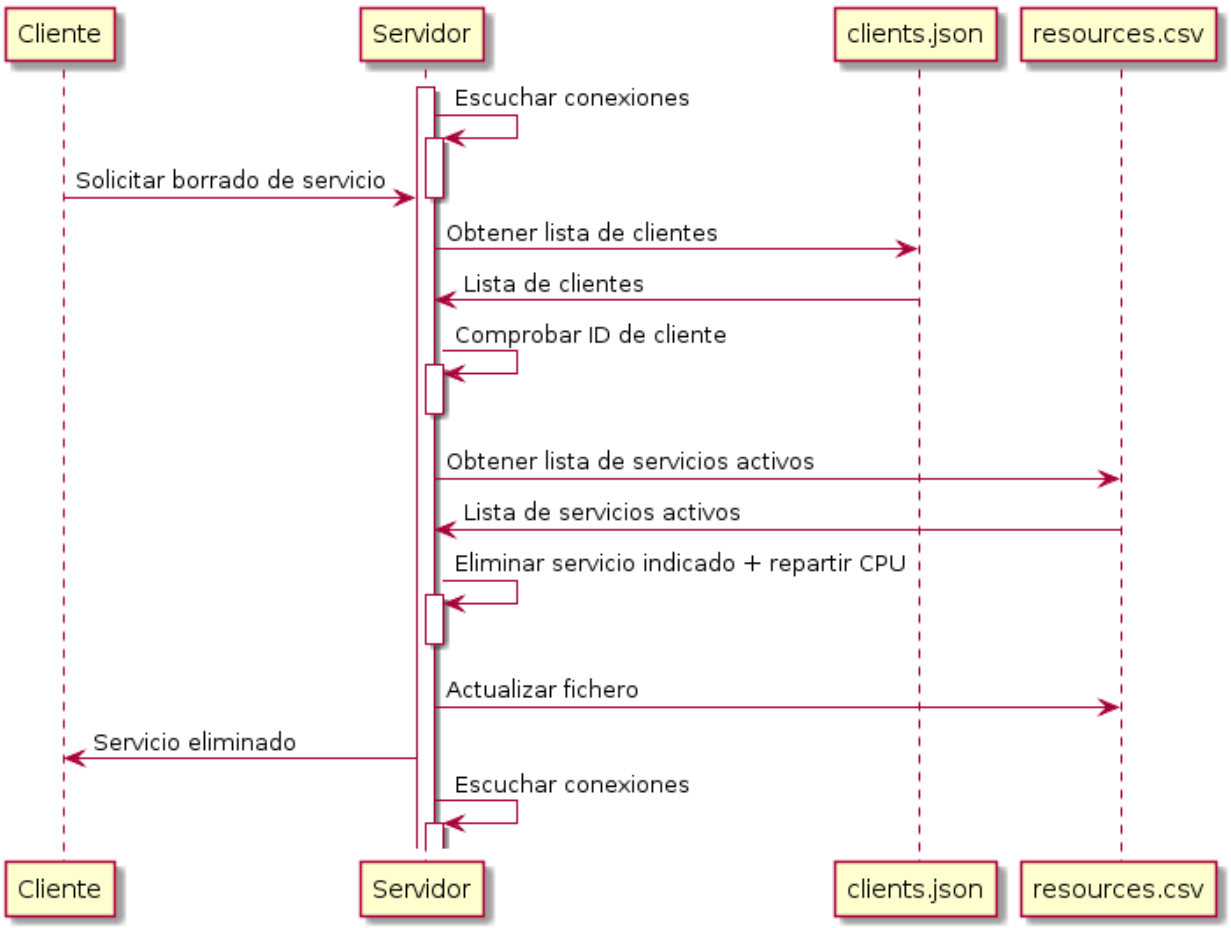


Figura 4.10: Diagrama de secuencia del servicio de eliminación de contenedor

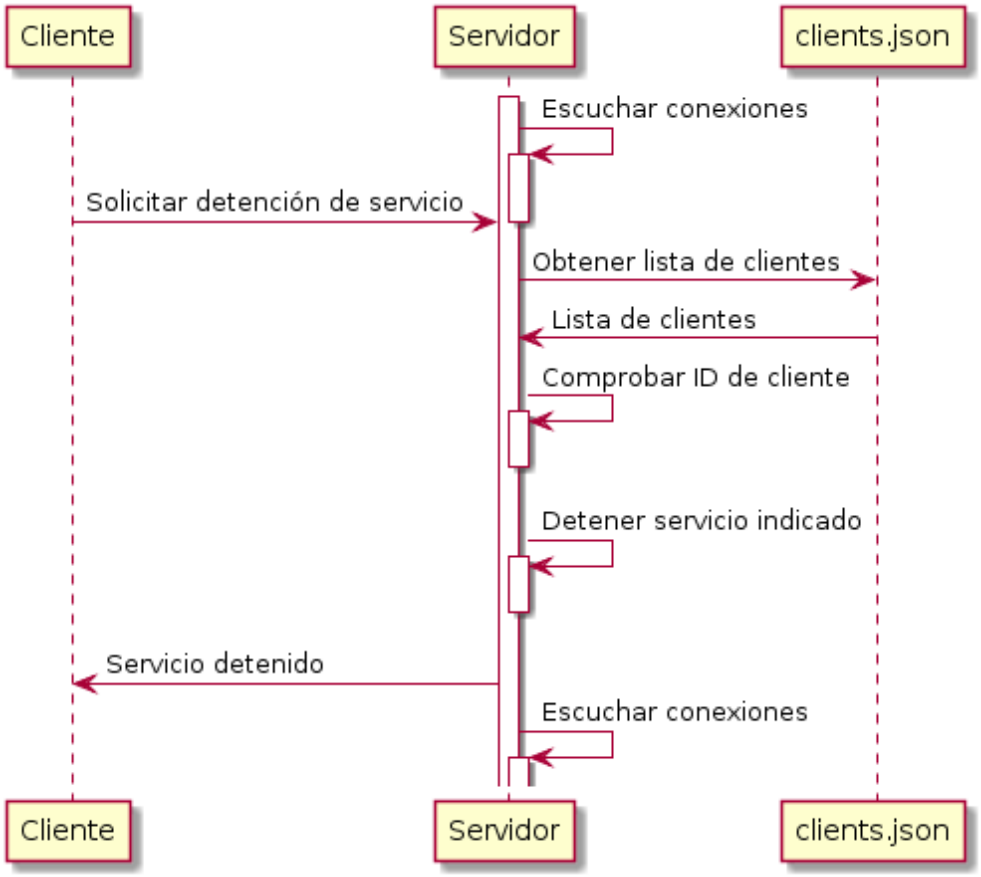


Figura 4.11: Diagrama de secuencia del servicio de detención de contenedor

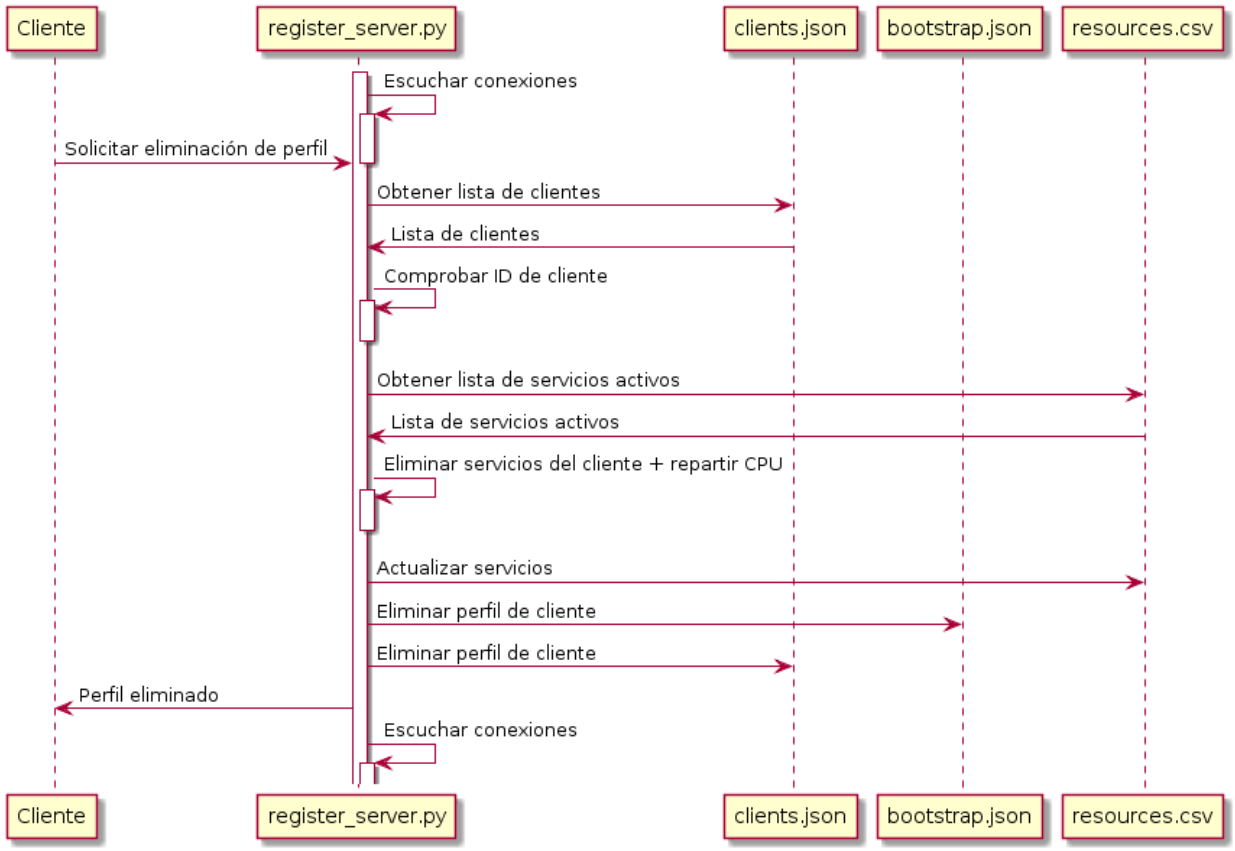


Figura 4.12: Diagrama de secuencia del servicio de eliminación de perfil

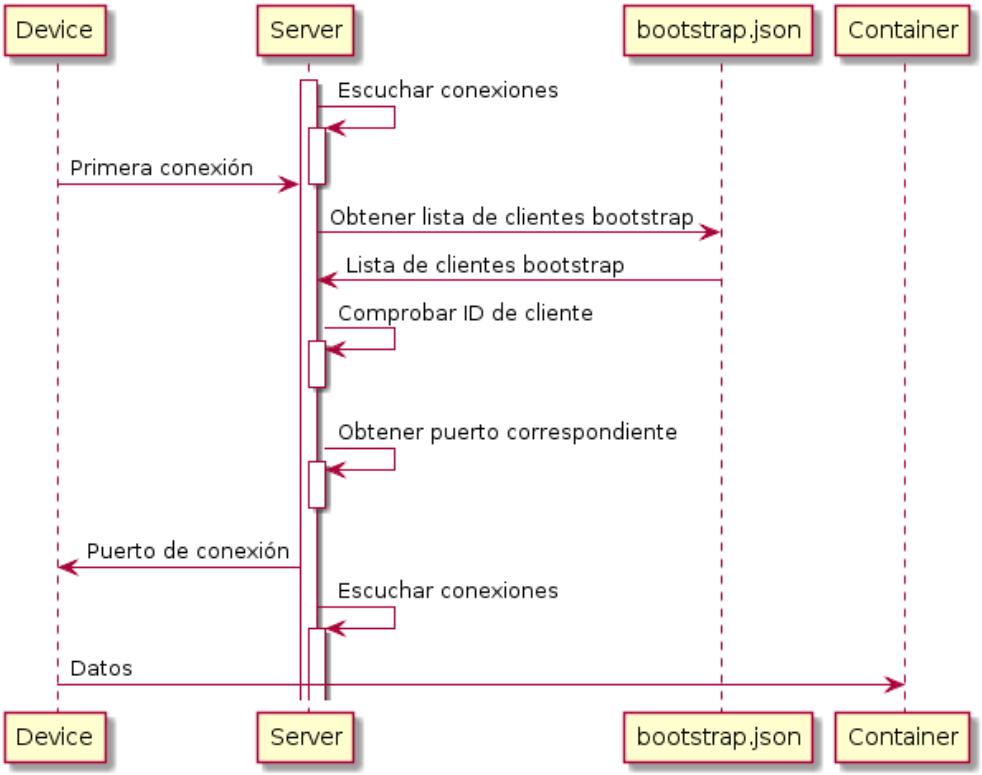


Figura 4.13: Diagrama de secuencia del servidor bootstrap

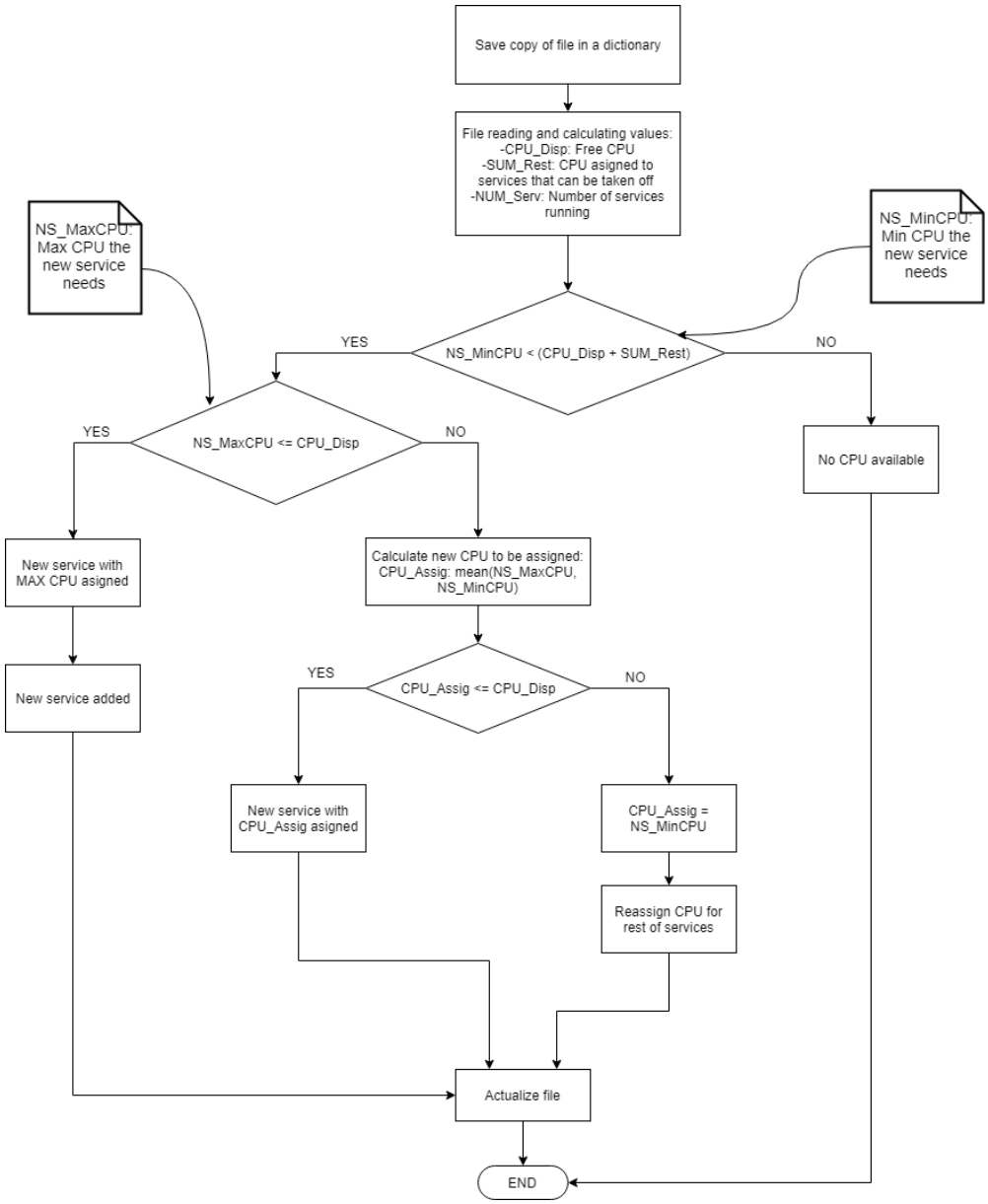


Figura 4.14: Algoritmo de gestión de recursos al levantar un servicio

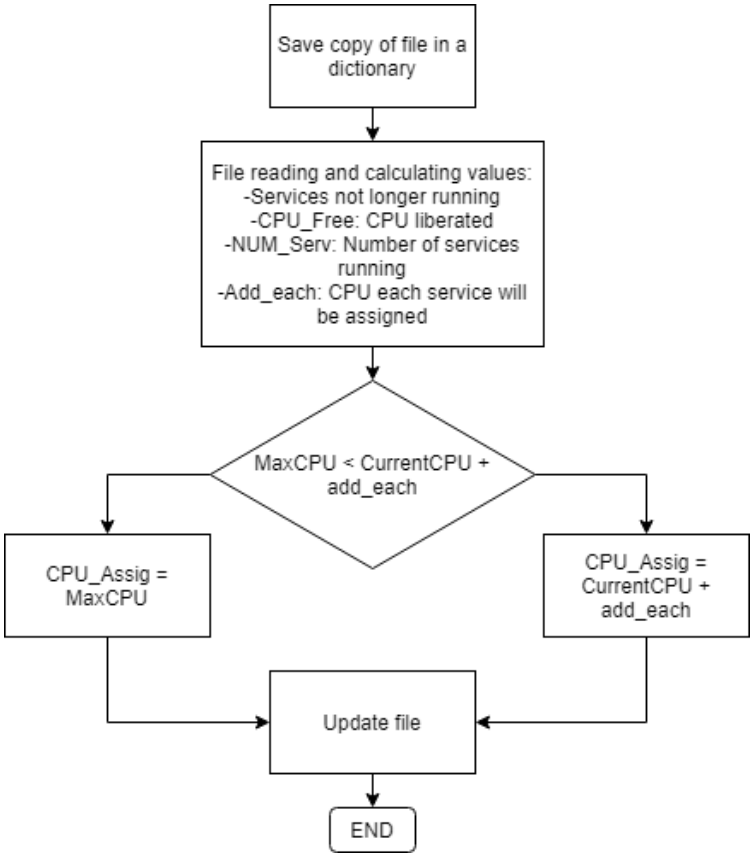


Figura 4.15: Algoritmo de gestión de recursos al eliminar servicios

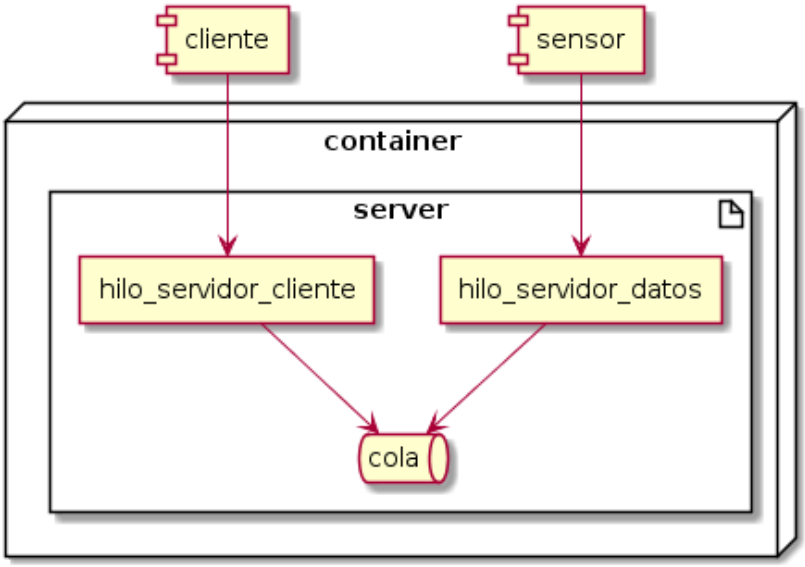


Figura 4.16: Arquitectura de los servicios

Capítulo 5

ECM. Evaluación de rendimiento

5.1. Valoración del funcionamiento del sistema

El principal objetivo del trabajo era demostrar que una máquina de prestaciones limitadas es capaz de sustentar un sistema que contenga la lógica necesaria para hacer funcionar un sistema *Edge Computing*.

Hasta ahora se ha mostrado la arquitectura de la solución, qué componentes participan y cómo interactúan entre ellos. A continuación, van a mostrarse los resultados de las pruebas realizadas, pruebas orientadas a comprobar que:

- El sistema es capaz de levantar contenedores bajo demanda, siendo robusto ante potenciales problemas de falta de espacio, puertos no disponibles, conexiones no autorizadas, etc.

- El sistema es capaz de dar servicio a más de un cliente de manera concurrente, ofreciendo capacidad de cómputo suficiente que permita ejecutar inferencias sobre los datos y proveer los resultados en plazos de tiempo adecuados. Se probará la capacidad de cómputo en distintas condiciones, variando una serie de parámetros que afectan al rendimiento de un contenedor.

    - Uno o más contenedores en paralelo realizando inferencia sobre datos.
    - Mayor o menor frecuencia de envío de datos.

- Mayor o menor tamaño del modelo analítico utilizado para realizar la inferencia.
- Mayor o menor número de *cores* de la CPU asignados al contenedor.
- Análisis sobre un único dato o en modo *batch*.

- El sistema es capaz de encapsular y aislar los servicios de tal manera que un cliente no pueda acceder a los datos de otro, y que en todas las conexiones los participantes se autentican correctamente.

5.1.1. Rendimiento de análisis en modo *batch*

Este primer bloque de pruebas persigue comprobar cuánto tarda un contenedor de análisis de datos en levantar el servicio y realizar inferencias sobre *batches* de datos de diferentes tamaños.

Se comprobará el tiempo que tarda en realizar las diferentes tareas necesarias para preparar el servicio -cargar las librerías y el modelo analítico-, y las inferencias -para distintos tamaños de *batch*-, bajo diferentes condiciones: contenedores en paralelo, asignación de *cores* y tamaño del modelo.

Prueba 1: Análisis en modo *batch* con único contenedor ejecutándose

La primera prueba consiste en medir el rendimiento del contenedor sin ningún otro ejecutándose paralelamente.

Se probará para dos modelos, uno de 74 KB y otro de 178 KB.

Los resultados de la prueba se aprecian en el cuadro 5.1.

Prueba 2: Análisis en modo *batch* con un contenedor ejecutándose en paralelo

La segunda prueba consiste en medir el rendimiento del contenedor con un contenedor ejecutándose en paralelo, realizando inferencias en modo dato a dato.

Los resultados de la prueba se aprecian en el cuadro 5.2.

Cores	**Carga modelo (s)**	**Carga librería (s)**	**50 datos (s)**	**100 datos (s)**	**500 datos (s)**
			Modelo de 74 KB		
2	16,2	15,13	0,56	0,56	0,58
1	18,26	16,36	0,6	0,61	0,59
0,5	37,82	35,4	0,96	1,49	1,42
0,25	99,5	96,7	2,94	2,94	2,06
0,1	252,6	247	7,71	7,71	8,243
			Modelo de 178 KB		
2	29,43	16,93	1,01	0,93	1,02
1	29,34	16,81	0,87	0,94	1,01
0,5	82,71	48,77	2,43	2,62	2,82
0,25	176,6	103,69	4,91	4,98	6,09
0,1	438,89	260,81	13,19	13,1	14,41

Cuadro 5.1: Resultados de análisis en batch, único contenedor con diferentes números de cores asignados

Cores	**Carga modelo (s)**	**Carga librería (s)**	**50 datos (s)**	**100 datos (s)**	**500 datos (s)**
			Modelo de 74 KB		
1,5-1,5	16,37	16,12	0,57	0,57	0,59
1,5-0,5	16,45	15,31	0,56	0,62	0,58
1-2	16,34	16,01	0,58	0,56	0,58
1-1	16,33	15,45	0,59	0,57	0,62
0,5-1,5	45,61	45,62	1,44	1,38	1,52
			Modelo de 178 KB		
1,5-1,5	29,4	15,98	1,04	0,98	0,97
1,5-0,5	29,41	16,02	1,12	1,02	0,99
1-2	29,44	16,77	1,09	0,97	1,2
1-1	29,39	16,13	0,98	1,02	1,14
0,5-1,5	82,17	46,61	2,52	2,64	2,55

Cuadro 5.2: Resultados de análisis en batch, un contenedor en paralelo con diferentes números de cores asignados

Prueba 3: Análisis en modo *batch* con dos contenedores ejecutándose en paralelo

La tercera prueba consiste en medir el rendimiento del contenedor con dos contenedores ejecutándose en paralelo, realizando inferencias en modo dato a dato.

Los resultados de la prueba se aprecian en el cuadro 5.3.

Cores	Carga modelo (s)	Carga librería (s)	50 datos (s)	100 datos (s)	500 datos (s)
Modelo de 74 KB					
1-1,5-1,5	16,29	15,22	0,57	0,58	0,67
1-0,5-0,5	16,41	15,92	0,61	0,59	0,65
0,5-0,75-0,75	45,35	46,52	1,41	1,34	1,49
0,3-0,3-0,3	83,69	81,19	2,46	2,49	2,79
0,1-0,9-0,9	254,51	249	7,67	7,8	8,33
Modelo de 178 KB					
1-1,5-1,5	27,39	15,25	0,96	0,88	0,96
1-0,5-0,5	29,97	16,44	0,97	1,22	1,14
0,5-0,75-0,75	82,23	47,03	2,65	2,58	2,63
0,3-0,3-0,3	135,9	81,7	4,04	4,12	4,22
0,1-0,9-0,9	440,92	253,79	13,22	14,2	14,15

Cuadro 5.3: Resultados de análisis en batch, dos contenedores en paralelo con diferentes números de cores asignados

Prueba 4: Análisis en modo *batch* con tres contenedores ejecutándose en paralelo

La cuarta prueba consiste en medir el rendimiento del contenedor con tres contenedores ejecutándose en paralelo, realizando inferencias en modo dato a dato.

Los resultados de la prueba se aprecian en el cuadro 5.4.

Prueba 5: Análisis en modo *batch* con cuatro contenedores ejecutándose en paralelo

La quinta prueba consiste en medir el rendimiento del contenedor con cuatro contenedores ejecutándose en paralelo, realizando inferencias en modo dato a dato.

Los resultados de la prueba se aprecian en el cuadro 5.5.

Cores	**Carga modelo (s)**	**Carga librería (s)**	**50 datos (s)**	**100 datos (s)**	**500 datos (s)**
Modelo de 74 KB					
1-1-1-1	16,92	16,8	0,6	0,61	0,66
1-0,3-0,3-0,3	16,26	15,34	0,63	0,59	0,66
0,5-1-1-1	45,1	44,66	1,39	1,43	1,45
0,5-0,5-0,5-0,5	45,41	44,66	1,39	1,43	1,45
Modelo de 178 KB					
1-1-1-1	29,73	16,06	1,16	1,04	1,14
1-0,3-0,3-0,3	30,02	15,94	0,92	1,05	1,02
0,5-1-1-1	75,11	45,09	2,17	2,35	2,53
0,5-0,5-0,5-0,5	83,14	45,88	2,66	2,61	2,7

Cuadro 5.4: Resultados de análisis en batch, tres contenedores en paralelo con diferentes números de cores asignados

Cores	**Carga modelo (s)**	**Carga librería (s)**	**50 datos (s)**	**100 datos (s)**	**500 datos (s)**
Modelo de 74 KB					
1-0,75-0,75-0,75-0,75	16,02	15,44	0,62	0,59	0,74
0,5-0,8-0,8-0,8-0,8	45,08	44,99	1,39	1,37	1,51
Modelo de 178 KB					
1-0,75-0,75-0,75-0,75	27,67	15,49	0,9	0,96	0,94
0,5-0,8-0,8-0,8-0,8	74,77	44,22	2,25	2,14	2,47

Cuadro 5.5: Resultados de análisis en batch, cuatro contenedores en paralelo con diferentes números de cores asignados

Conclusiones del análisis de rendimiento en modo *batch*

A la vista de los resultados obtenidos en las pruebas realizadas sobre el servidor de análisis en modo *batch*, las conclusiones son las siguientes:

- El tamaño del modelo afecta a la velocidad con la que se provee el servicio. En la figura 5.1 se puede comprobar cómo varía, para cada asignación de *cores*, en función del tamaño del modelo.

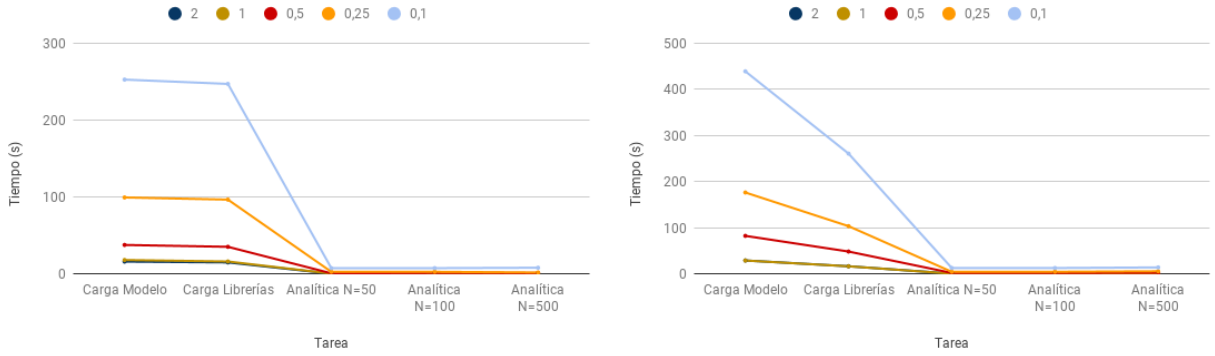
- Lo que más limita la velocidad con la que el servicio se despliega es la carga del modelo analítico y de las librerías. El tiempo invertido en estas dos tareas, como puede apreciarse en la figura 5.2, se dispara para asignaciones de *cores* muy bajas -por debajo de 0,5-, superando incluso los 2 o 3 minutos. En el caso de las inferencias, el tiempo necesario para realizarlas es suficientemente rápido salvo en los casos más extremos, en los que la CPU asignada se limita a 0,1.

- Que haya contenedores ejecutándose en paralelo no afecta al rendimiento del análisis, incluso aunque estén realizando tareas de inferencias que requieran más capacidad de cómputo que otras más simples. Los tiempos requeridos para cada tarea apenas varían en cada caso, por lo que se puede concluir que los únicos factores que afectan al rendimiento de la inferencia son el tamaño del modelo y la CPU asignada al contenedor.

- A partir de 1 *core* asignado no hay diferencia de rendimiento, por lo que de cara a configurar el servicio de gestión de recursos, se limitará a este valor los *cores* máximos asignables a un contenedor. Así será posible dar servicio a más clientes de manera concurrente.

5.1.2. Rendimiento de análisis en modo dato a dato

Este segundo bloque de pruebas persigue comprobar cuánto tarda un contenedor de análisis de datos en realizar una inferencia sobre un único dato que llegue proveniente de los sensores.



(a) Modelo de 78 KB

(b) Modelo de 174 KB

Figura 5.1: Comparación de rendimiento de análisis para diferentes números de cores asignados y modelos

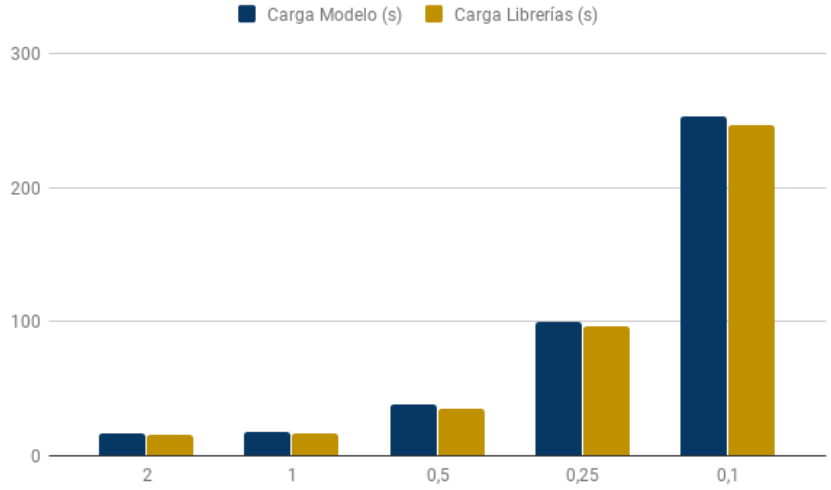


Figura 5.2: Tiempo empleado en la carga del modelo analítico y de las librerías

Al igual que se hizo en el anterior bloque de pruebas, se probará dicha capacidad para diferentes asignaciones de *cores*, con uno o varios contenedores corriendo en paralelo y para dos modelos analíticos de distintos tamaños.

Prueba 1: Análisis en modo dato a dato con único contenedor ejecutándose

La primera prueba consiste en medir el rendimiento del contenedor sin ningún otro ejecutándose paralelamente.

Se probará para dos modelos, uno de 74 KB y otro de 178 KB.

Los resultados se aprecian en el cuadro 5.6.

Cores	**Analítica 1er dato (s)**	**Siguientes analíticas (s)**			
Modelo de 74 KB					
2	0,53	0,007	0,018	0,008	0,009
1	0,65	0,089	0,082	0,082	0,084
0,75	0,65	0,039	0,006	0,032	0,01
0,5	1,16	0,025	0,023	0,007	0,016
0,25	2,89	0,015	0,008	0,01	0,056
0,1	7,49	0,007	0,07	0,09	0,006
Modelo de 178 KB					
2	0,77	0,014	0,011	0,01	0,012
1	0,78	0,0011	0,01	0,013	0,012
0,75	1,17	0,016	0,011	0,08	0,07
0,5	2,49	0,01	0,04	0,0011	0,012
0,25	5,39	0,09	0,07	0,06	0,04
0,1	13,79	0,11	0,09	0,08	0,06

Cuadro 5.6: Resultados de análisis en batch, único contenedor en paralelo con diferentes números de cores asignados

Prueba 2: Análisis en modo dato a dato con un contenedor ejecutándose en paralelo

La segunda prueba consiste en medir el rendimiento del contenedor con un contenedor ejecutándose paralelamente.

Los resultados se aprecian en el cuadro 5.7.

Prueba 3: Análisis en modo dato a dato con dos contenedores ejecutándose en paralelo

La tercera prueba consiste en medir el rendimiento del contenedor con dos contenedores ejecutándose paralelamente.

Los resultados se aprecian en el cuadro 5.8.

Cores	**Analítica 1er dato (s)**	**Siguientes analíticas (s)**			
		Modelo de 74 KB			
1,5-1,5	0,55	0,044	0,009	0,071	0,023
1,5-0,5	0,57	0,022	0,04	0,025	0,033
1-2	0,54	0,008	0,022	0,087	0,012
1-1	0,59	0,056	0,035	0,056	0,044
0,5-1,5	1,22	0,049	0,067	0,011	0,091
		Modelo de 178 KB			
1,5-1,5	0,78	0,023	0,069	0,085	0,095
1,5-0,5	0,69	0,092	0,063	0,095	0,09
1-2	0,77	0,003	0,063	0,044	0,06
1-1	0,74	0,083	0,031	0,054	0,011
0,5-1,5	2,55	0,011	0,019	0,033	0,026

Cuadro 5.7: Resultados de análisis en batch, un contenedor en paralelo con diferentes números de cores asignados

Cores	**Analítica 1er dato (s)**	**Siguientes analíticas (s)**			
		Modelo de 74 KB			
1-1,5-1,5	0,47	0,03	0,007	0,023	0,009
1-0,5-0,5	0,63	0,005	0,045	0,008	0,037
0,5-0,75-0,75	1,11	0,52	0,084	0,056	0,06
0,3-0,3-0,3	2,36	0,048	0,007	0,007	0,032
0,1-0,9-0,9	7,38	0,023	0,055	0,042	0,043
		Modelo de 178 KB			
1-1,5-1,5	0,73	0,027	0,01	0,023	0,009
1-0,5-0,5	0,75	0,035	0,01	0,045	0,038
0,5-0,75-0,75	2,47	0,043	0,084	0,038	0,055
0,3-0,3-0,3	3,88	0,007	0,047	0,041	0,038
0,1-0,9-0,9	14,07	0,04	0,09	0,013	0,012

Cuadro 5.8: Resultados de análisis en batch, dos contenedores en paralelo con diferentes números de cores asignados

Prueba 4: Análisis en modo dato a dato con tres contenedores ejecutándose en paralelo

La cuarta prueba consiste en medir el rendimiento del contenedor con tres contenedores ejecutándose paralelamente.

Los resultados se aprecian en el cuadro 5.9.

Cores	**Analítica 1er dato (s)**	**Siguientes analíticas (s)**			
Modelo de 74 KB					
1,1,1,1	0,71	0,032	0,066	0,038	0,083
1-0,3-0,3-0,3	0,6	0,1	0,087	0,013	0,039
0,5-1-1-1	1,19	0,003	0,037	0,064	0,013
0,5-0,5-0,5-0,5	1,31	0,02	0,008	0,1	0,086
Modelo de 178 KB					
1,1,1,1	0,68	0,048	0,068	0,095	0,005
1-0,3-0,3-0,3	0,77	0,041	0,026	0,065	0,039
0,5-1-1-1	2,03	0,011	0,008	0,023	0,25
0,5-0,5-0,5-0,5	2,48	0,004	0,008	0,023	0,025

Cuadro 5.9: Resultados de análisis en batch, tres contenedores en paralelo con diferentes números de cores asignados

Prueba 5: Análisis en modo dato a dato con cuatro contenedores ejecutándose en paralelo

La quinta prueba consiste en medir el rendimiento del contenedor con cuatro contenedores ejecutándose paralelamente.

Los resultados se aprecian en el cuadro 5.10.

Prueba 6: Frecuencia máxima soportable por el servicio

Esta sexta prueba consiste en comprobar cuál es la frecuencia máxima de envío que puede soportar el servicio, con diferentes asignaciones de *cores* y para los dos modelos utilizados hasta el momento.

Cores	Analítica 1er dato (s)	Siguientes analíticas (s)			
Modelo de 74 KB					
1,0,75-0,75-0,75-0,75	0,48	0,009	0,086	0,007	0,008
0,5-0,8-0,8-0,8-0,8	1,21	0,009	0,2	0,009	0,007
Modelo de 178 KB					
1,0,75-0,75-0,75-0,75	0,77	0,038	0,012	0,008	0,01
0,5-0,8-0,8-0,8-0,8	2,05	0,011	0,009	0,09	0,1

Cuadro 5.10: Resultados de análisis en batch, cuatro contenedores en paralelo con diferentes números de cores asignados asignados

Los resultados de la prueba se aprecian en el cuadro 5.11.

Conclusiones del análisis de rendimiento en modo dato a dato

A la vista de los resultados obtenidos en las pruebas realizadas sobre el servidor de análisis en modo dato a dato, las conclusiones son las siguientes:

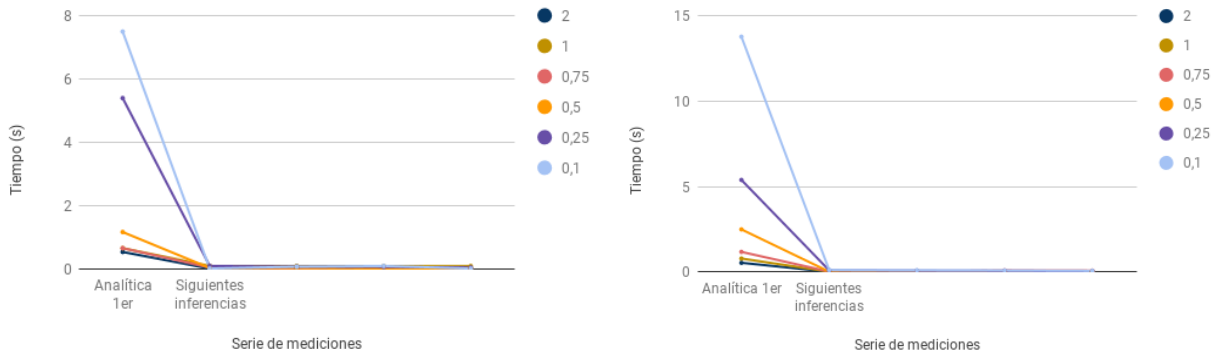
- La primera inferencia realizada tarda significativamente más que las siguientes, como puede apreciarse en la figura 5.3. Es este tiempo de inferencia el que limita la mayoría de las frecuencias asignables.
- El resto de inferencias apenas varían, tanto dentro de una misma ejecución como para distintos *cores* asignados y distintos tamaños de modelo. Existen algunos picos que también limitan la frecuencia, pero no se deben a los *cores* asignados sino a determinados estados del planificador del sistema.

5.1.3. Tiempo total de provisión de servicio

Este tercer bloque de pruebas persigue medir el tiempo total que necesita el sistema en recibir una conexión de un cliente y ejecutar la tarea solicitada.

Cores	**Frecuencia límite (s)**
Modelo de 74 KB	
2	0,3
1	0,3
0,75	0,4
0,5	0,7
0,25	1,5
0,1	4
Modelo de 178 KB	
2	0,5
1	0,5
0,75	0,6
0,5	1,2
0,25	2,7
0,1	6,6

Cuadro 5.11: Resultados de análisis de la frecuencia máxima de envío de datos soportada



(a) Modelo de 78 KB

(b) Modelo de 174 KB

Figura 5.3: Comparación de rendimiento de análisis dato a dato para diferentes números de cores asignados y modelos

Se medirán los tiempos de respuesta para cada servicio, en condiciones de no existir contenedores levantados, o de haber uno o varios ejecutándose.

Además, también se comprobará la diferencia de tiempo entre levantar un contenedor de cero y arrancar uno que se encuentre detenido.

Prueba 1: Provisión de servicios sin contenedores ejecutándose en paralelo

La primera prueba consiste en medir el tiempo de respuesta de cada servicio sin haber ningún contenedor ejecutándose.

Los resultados de la prueba se aprecian en el cuadro 5.12.

Servicio	**Tiempo (s)**		
Datos	9,16	7,81	6,42
Análisis	10,16	9,11	10,93
Eliminar servicio	4,1	6,54	4,65
Detener servicio	4,5	6,24	4,68
Eliminar perfil	0,74	0,89	0,29
Información	0,034	0,026	0,028

Cuadro 5.12: Resultados de análisis de tiempo de provisión de servicio sin contenedor en paralelo

Prueba 2: Provisión de servicios con un contenedor ejecutándose en paralelo

La segunda prueba consiste en medir el tiempo de respuesta de cada servicio habiendo un contenedor ejecutándose.

Los resultados de la prueba se aprecian en el cuadro 5.13.

Servicio	**Tiempo (s)**		
Datos	8,23	6,36	6,37
Análisis	8,09	11,44	9,01
Eliminar servicio	7,3	6,92	5,89
Detener servicio	4,41	5,38	5,41
Eliminar perfil	2,04	0,89	0,29
Información	0,028	0,025	0,025

Cuadro 5.13: Resultados de análisis de tiempo de provisión de servicio con un contenedor en paralelo

Prueba 3: Provisión de servicios con dos contenedores ejecutándose en paralelo

La tercera prueba consiste en medir el tiempo de respuesta de cada servicio habiendo dos contenedores ejecutándose.

Los resultados de la prueba se aprecian en el cuadro 5.14.

Servicio	**Tiempo (s)**		
Datos	6,04	6,28	7,79
Análisis	9,81	9,26	12,88
Eliminar servicio	7,77	7,91	7,79
Detener servicio	5,1	4,85	4,11
Eliminar perfil	3,17	3,58	3,23
Información	0,026	0,027	0,03

Cuadro 5.14: Resultados de análisis de tiempo de provisión de servicio con dos contenedores en paralelo.

Prueba 4: Comparación de tiempos entre levantar y arrancar contenedor

La cuarta prueba consiste en comparar los tiempos medidos para dos situaciones distintas: cuando un servidor levantad desde cero un contenedor partiendo de una imagen base, y cuando arranca un contenedor que se encuentra detenido.

Los resultados de la prueba se aprecian en el cuadro 5.15.

Servicio	**Tiempo (s)**		
Ningún contenedor en paralelo			
Datos-levantar	9,16	7,81	6,42
Datos-arranque	6,78	6,03	5,5
Análisis-levantar	10,16	9,11	10,93
Análisis-arranque	9,61	9,8	9,46
Un contenedor en paralelo			
Datos-levantar	8,23	6,36	6,37
Datos-arranque	5,36	5,53	7,47
Análisis-levantar	8,09	11,44	9,01
Análisis-arranque	9,05	10,81	8,59
Dos contenedores en paralelo			
Datos-levantar	6,04	6,28	7,79
Datos-arranque	7,25	8,43	7,11
Análisis-levantar	9,81	9,26	12,88
Análisis-arranque	10,01	11,22	12,53

Cuadro 5.15: Resultados de análisis de tiempo de provisión de servicio con dos contenedores en paralelo.

Conlusiones del análisis del tiempo de provisión de servicio

A la vista de los resultados obtenidos en las pruebas de medición de tiempos de provisión de servicio, las conclusiones son las siguientes:

- Levantar el servicio de analítica de datos tarda significativamente más que levantar el servicio de redirección de datos. Esto se debe a que la imagen del servidor de analítica es bastantes más pesada -998 MB- que la del servidor de datos -207 MB-. Esta diferencia viene producida, entre otras cosas, por las librerías de *Tensorflow* instaladas en la imagen. Las diferencias, bajo diferentes condiciones, pueden apreciarse en la figura 5.4

- La diferencia entre detener la ejecución de un contenedor y su eliminación apenas existe cuando no hay contenedores ejecutándose, pero se incrementa conforme las ejecuciones los incluyen -hasta alcanzar diferencias de dos y tres segundos-. Este hecho se puede apreciar en la figura 5.5. Se debe a la ejecución del algoritmo que reparte la CPU que deja libre el contenedor al ser eliminado; cuantos más contenedores haya ejecutándose, más se alarga el proceso de reparto. Por su parte, el servicio de detención se mantiene constante puesto que no precisa de interactuar con otros contenedores.

- Que el número de contenedores ejecutándose afecte al tiempo de provisión también sucede con el servicio de eliminación de perfil. La figura 5.6 muestra que el tiempo de ejecución de la tarea aumenta considerablemente según hay más contenedores ejecutándose. Esto se debe a la misma razón que en el punto anterior: el algoritmo que reparte los *cores* que han quedado libres gana complejidad cuanto más servicios haya levantados.

- El tiempo que tarda en proveerse el servicio de eliminado de perfil también escala según cuántos servicios tenga levantados el cliente que lo solicita. En la figura 5.6 se puede comprobar cómo aumenta dicho valor.

5.1.4. Capacidad máxima de la plataforma

Esta cuarta prueba persigue comprobar cuál es la capacidad máxima de la plataforma respecto al consumo de recursos del sistema.

El dispositivo de borde tiene una CPU de 4 *cores*, y Docker permite asignar hasta dos *cores* por contenedor. El objetivo de esta prueba consiste en ver qué sucede si se supera la asignación de *cores* disponibles.

La prueba consistirá en levantar una serie de contenedores y ejecutar tareas pesadas en ellos, como inferencias sobre datos. Se espera que, llegado un punto en el que la asignación de *cores* supera el número total que tiene la máquina, esta deja de funcionar.

Prueba con servicios de ECM

La primera prueba consiste en comprobar si el sistema se ve saturado ante el múltiple levantamiento de contenedores que ejecuten servicios desarrollados para este sistema.

La figura 5.7 muestra cuánta CPU consumen una serie de contenedores levantados, cuya información puede consultarse en el cuadro 5.16.

ID contenedor	Tipo de servicio	CPU asignada	Tarea ejecutándose
2872	análisis	1	inferencia
1654	datos	1	envío
214	análisis	1	carga módulos y librerías

Cuadro 5.16: Resultados de análisis de tiempo de provisión de servicio sin contenedor en paralelo

Como puede apreciarse, el consumo de CPU de estos servicios es verdaderamente bajo; una caída del sistema necesitaría la ejecución de una gran cantidad de contenedores.

Prueba con imágenes de terceros

La primera prueba consiste en comprobar si el sistema se ve saturado ante el múltiple levantamiento de contenedores que ejecuten servicios desarrollados por terceros.

En este caso, se utilizará un contenedor con la librería *Tensorflow* instalada para comprobar cuánta CPU consume mientras ejecuta diferentes etapas de un proceso de análisis de imágenes. El algoritmo de dicha tarea se obtuvo de un tutorial de *Keras* [28].

Estas etapas son consultables en el cuadro 5.17. Mencionar que la asignación de *cores* será de dos.

Los diferentes consumos de CPU puede apreciarse en la figura 5.8.

Etapa	**Consumo de CPU**
Descarga de datos	103,40 %
Entrenamiento del modelo	149,10 %
Inferencia de datos	128,77 %

Cuadro 5.17: Resultados de análisis de tiempo de provisión de servicio sin contenedor en paralelo

Como puede apreciarse, el consumo de CPU de este servicio es notablemente superior al de los servicios ECM. Por lo tanto, muchos menos contenedores son necesarios para saturar el sistema.

Se probó seguidamente a levantar dos contenedores que ejecutasen el algoritmo anterior, y la CPU se repartió como se aprecia en la figura 5.9.

Cuando ambos contenedores pasaron a la tarea de entrenamiento del modelo, el sistema cayó.

Conclusiones del análisis de la capacidad máxima del sistema

A la vista de los resultados obtenidos, las conclusiones son las siguientes:

- Los servicios desarrollados por defecto para ECM no suponen una amenaza para la robustez del sistema, pues no consumen cantidades de CPU suficientemente altas, y el algoritmo de gestión de recursos termina por cerrar toda posibilidad de darse esta situación.

- Otros análisis sobre datos más pesados sí consumen cantidades de CPU que provocan

que el algoritmo de gestión de recursos sea indispensable, para limitar el consumo de estos.

- La asignación de un número determinado de *cores* a un contenedor no supone que este contenedor los posea en exclusividad, sino que los comparte con el resto, así como con el sistema anfitrión. Esta asignación limita el número de *cores* máximos que puede consumir.

- El consumo de CPU permite cierta flexibilidad, que se traduce en que la CPU puede verse sobrecargada hasta un límite.

- Docker no realiza gestión de recursos, por lo que en situaciones en las que el consumo de CPU se dispara por parte de los contenedores, el sistema cae.

5.1.5. Aislamiento de contenedores

Esta quinta prueba persigue comprobar que el sistema es capaz de aislar los contenedores y las comunicaciones de los clientes, de tal manera que cada flujo de datos queda accesible únicamente al cliente dueño de los sensores que están generando estos datos.

Se han provisto dos mecanismos para alcanzar este grado de aislamiento:

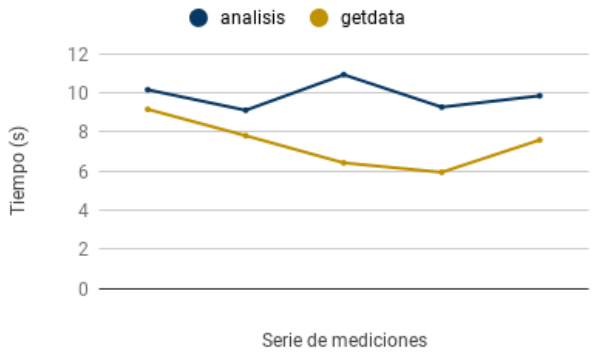
- Mapeo de puertos: Docker permite, al levantar un contenedor, mapear puertos de la máquina anfitrión con los del propio contenedor. Así, toda comunicación dirigida a un puerto de la máquina será automáticamente rederigida al puerto del servicio.

- Cadena de conexión: Cuando un cliente se registra, recibe una cadena de conexión única, que han de proveer tanto los sensores cuando envíen datos, como el propio cliente en cada conexión que realice con la plataforma.

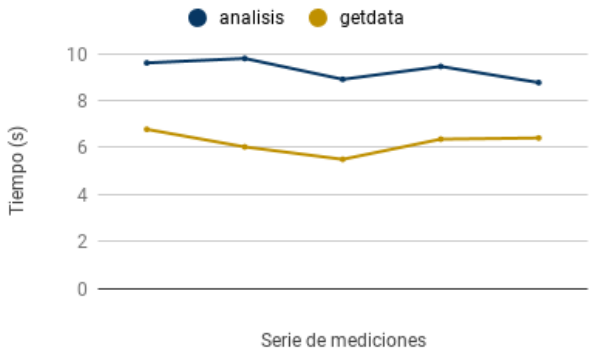
Como se ha podido observar en el capítulo 3, para todas y cada una de las comunicaciones con la plataforma, el cliente provee al sistema de la cave de conexión -menos en el servicio de registro-.

A lo largo de todos los ejemplos ilustrados puede observarse que la cadena es correcta, puesto que todas las solicitudes reciben mensajes afirmando que el servicio u operación solicitados han sido ejecutados con éxito.

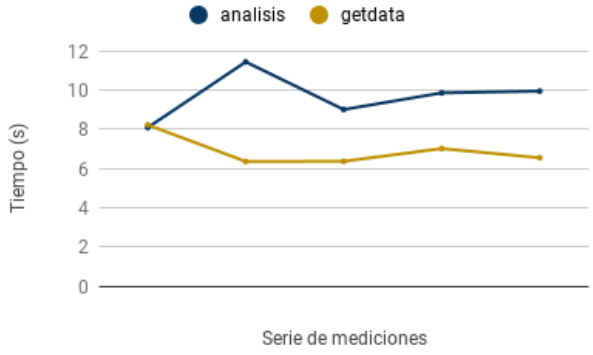
Sin embargo, como se aprecia en la figura 5.10, en caso de proveer una cadena de conexión incorrecta, el sistema devuelve un mensaje de error, y la comunicación finaliza sin haberse ejecutado el servicio solicitado.



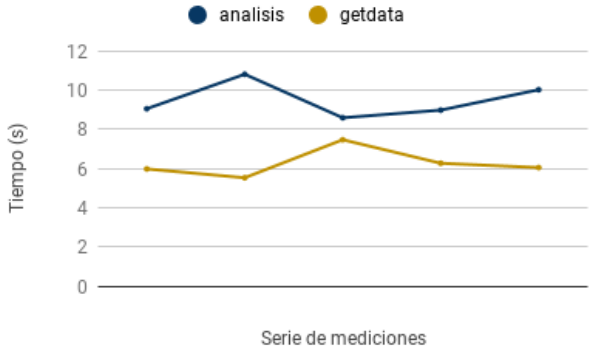
(a) Levantamiento con cero contenedores



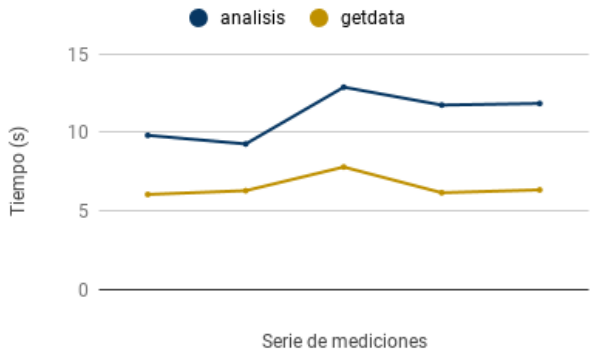
(b) Arranque con cero contenedores



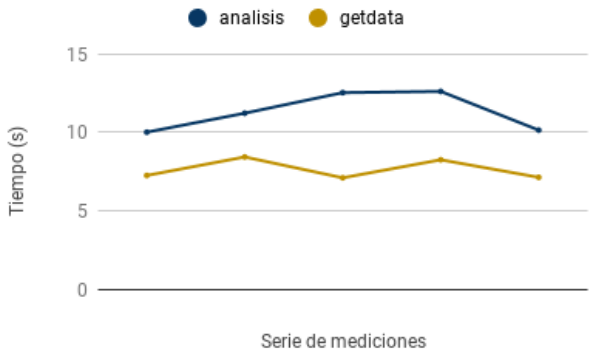
(c) Levantamiento con un contenedor



(d) Arranque con un contenedor

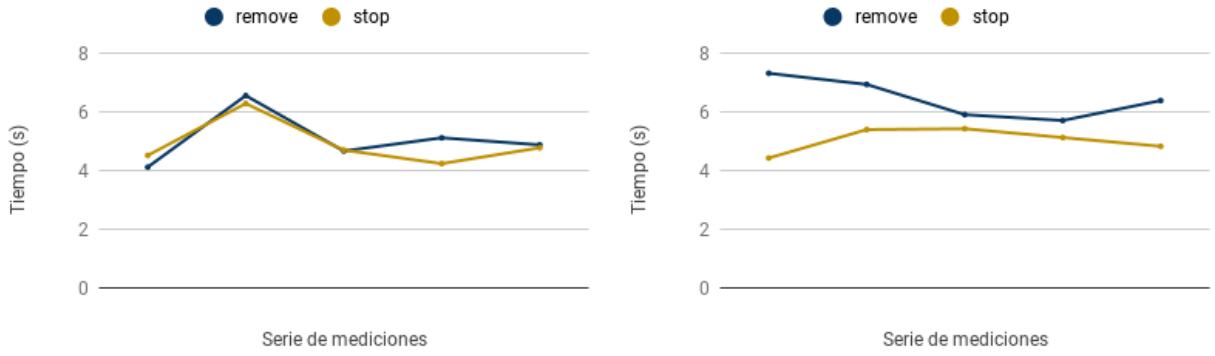


(e) Levantamiento con dos contenedores



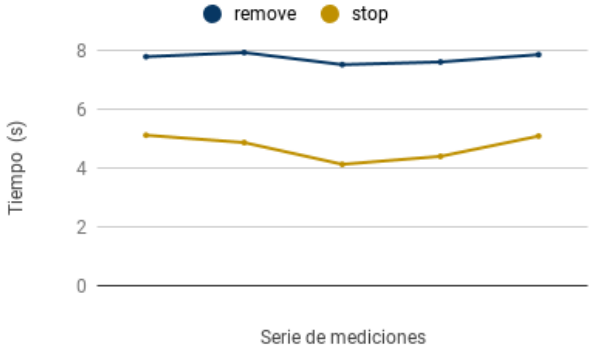
(f) Arranque con dos contenedores

Figura 5.4: Comparación de tiempos de provisión de servicios bajo diferentes condiciones



(a) Cero contenedores en paralelo

(b) Un contenedor en paralelo



(c) Dos contenedores en paralelo

Figura 5.5: Comparación de rendimiento entre los servicios de borrado y detención

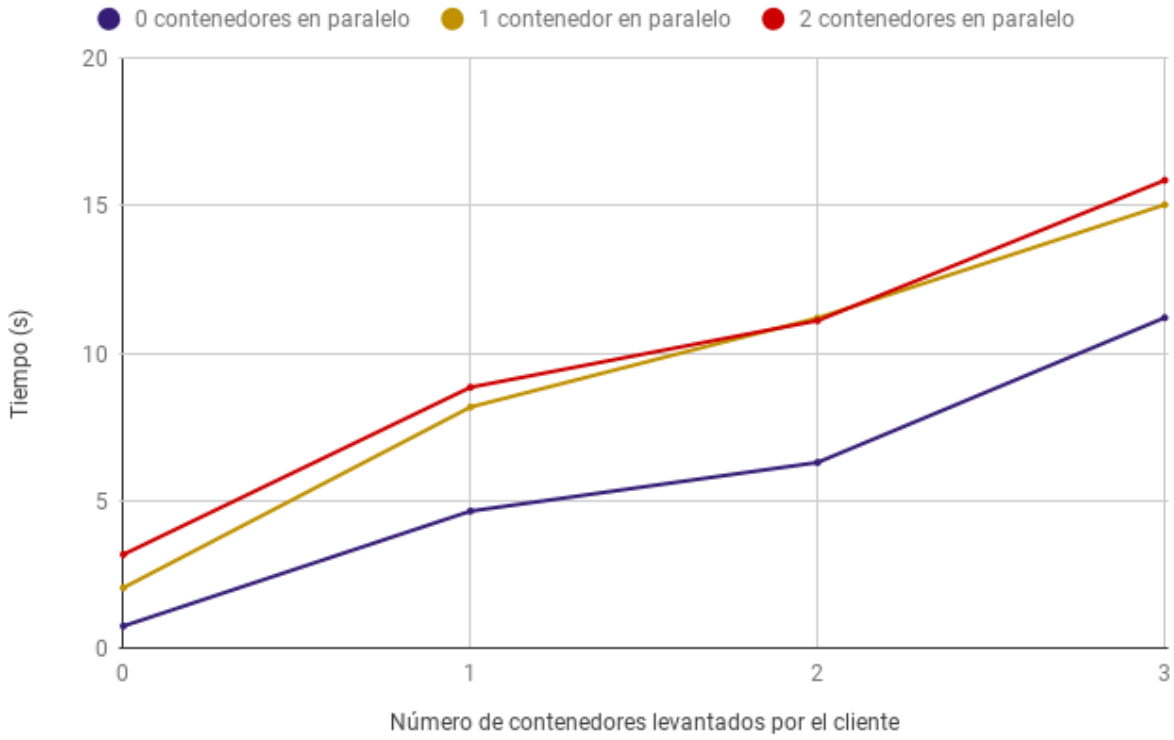


Figura 5.6: Comparación de rendimiento del servicio de eliminado de perfil para distino número de contenedores ejecutándose en paralelo, y para distintos contenedores levantados por el usuario

```
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O           PIDS
ea9cd57651d2        2872                0.87%               78.84MiB / 1001MiB    7.88%               7.63kB / 2.8kB      107MB / 0B          0
bd94956295fc        1654                0.16%               4.203MiB / 1001MiB    0.42%               43.6kB / 16.4kB     10.8MB / 0B         0
eb8da424bfe4        214                 52.52%              69.6MiB / 1001MiB     6.95%               648B / 0B           0B / 0B             0
```

Figura 5.7: Consumo de CPU por parte de servicios ECM

```
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O           PIDS
a2ac28af9bf1        4122                103.40%             60.07MiB / 1001MiB    6.00%               648B / 0B           918kB / 4.1kB       0
```

(a) Consumo de CPU al descargar datos

```
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O           PIDS
a2ac28af9bf1        4122                149.10%             492.4MiB / 1001MiB    49.19%              31.5MB / 212kB      10.2MB / 8.19kB     0
```

(b) Consumo de CPU al entrenar modelo

```
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O           PIDS
a2ac28af9bf1        4122                128.77%             493.5MiB / 1001MiB    49.30%              31.5MB / 212kB      10.2MB / 8.19kB     0
```

(c) Consumo de CPU al realizar inferencia

Figura 5.8: Consumo de CPU por parte de servicio externo

```
CONTAINER ID   NAME      CPU %     MEM USAGE / LIMIT   MEM %     NET I/O         BLOCK I/O       PIDS
87d2cf0b18a0   424       84.50%    62.56MiB / 1001MiB  6.25%     30.3MB / 206kB  3.58MB / 4.1kB  0
a2ac28af9bf1   4122      87.96%    105.8MiB / 1001MiB  10.57%    31.5MB / 212kB  10.2MB / 8.19kB 0
```

Figura 5.9: Consumo de CPU repartida entre varios servicios externos

```
root@lamobo-r1:~/client_tcp# python3 cliente.py -s listening --id clavenoexistente
{"number_values": 10, "lports": ["default"], "service": {"default_service": "listening", "custom": 0, "image": "null"}, "cpus": {"max": 1, "min": 0.5}, "data": "./data/temp.csv", "nports": 1,
 "order": "null", "batch": 0, "serviceid": 0, "id": "clavenoexistente"}
Now waiting for response
Received resp: b'{"body": "El ID provisto no corresponde con ningun cliente registrado", "status": "ERROR"}'
Tiempo de operacion: 0.0272414684295654 3
```

(a) Error al solicitar servicio

```
root@lamobo-r1:~/client_tcp# python3 container_connection.py -p 5031 --id clavenoexistente
ERROR. Conexion no permitida
```

(b) Error al conectarse a contenedor

Figura 5.10: Mensajes de error al introducir clave de conexión errónea

Capítulo 6

Conclusiones

El principal objetivo del proyecto era demostrar que un dispositivo de prestaciones limitadas podía ser utilizado como dispositivo de borde para una arquitectura *Edge Computing*.

A lo largo de la memoria se ha ido representando la construcción de una arquitectura inspirada en una casa conectada, en la que diversos sensores proporcionan datos que eran analizados en el propio *gateway*.

La plataforma ECM está construida para que el análisis de los datos se realice bajo demanda, siendo un ciente el que solicita que se levante un servicio que ejecute dicha tarea.

Otro de los objetivos del trabajo era automatizar toda la gestión del servicio, que requería los siguientes controles:

- Gestión de los servidores que levantan los contenedores.
- Gestión de los recursos asignados a cada contenedor, de tal manera que el dispositivo no se sobrecargase.
- Gestión de los clientes y de las solicitudes de altas y bajas.

Todos los servicios deben ofrecer un mínimo de capacidad de cómputo y de eficiencia, de tal manera que el tiempo transcurrido entre que el cliente solicita un servicio y este es levantado fuese mínimo, y que las tareas de inferencia de datos también se ejecutasen de manera suficientemente rápida.

Además, los servicios debían quedar aislados unos de otros, para asegurar la protección de los datos de los clientes.

Por tanto, hay que concluir que:

- **La capacidad de cómputo del dispositivo de borde es suficientemente alta:** El dispositivo utilizado soporta tareas de análisis de datos cuando el tamaño de estos no es excesivamente grande. En en entorno como el propuesto en este proyecto, donde los datos son valores de las condiciones del entorno, la capacidad de cómputo es suficiente para realizar inferencias sobre los datos.

- **El número de servicios que pueden levantarse simultáneamente es limitado:** Pese a que el dispositivo es capaz de procesar y analizar datos, incluso cuando la frecuencia de envío de estos es alta, el número de servicios que pueden tenerse levantados es limitado. Esta restricción se debe al número de *cores* que tiene el dispositivo y, que para asignaciones por debajo de un *core*, el rendimiento se ve notablemente resentido.

- **La plataforma es robusta ante altos picos de trabajo:** Cuando el número de solicitudes es alta, y ejecutan en mayor medida tareas de análisis de datos, el sistema es capaz de soportar estas altas cargas de trabajo. Tanto el algoritmo de gestión de recursos como el propio Docker aportan funcionalidades que dotan a la máquina de dicha robustez.

- **El funcionamiento de la plataforma es autónomo:** ECM está desarrollado de tal manera que los contenedores son levantados de manera automática según se solicitan los servicios, y la gestión de recursos no requiere de intervención humana pues las asignaciones de los mismos se realizan de manera dinámica, respondiendo a las circunstancias del sistema.

Capítulo 7

Introduction

7.1. Motivation

The evolution of the Internet of Things (IoT) is so that, by the year 2024, it is expected to be 4 billion devices connected to Internet, only in IoT environments. If we add to that value the amount of other kind of connected devices such as smartphones, the expected value increases to 8.9 billion.

This amount of connected devices threats traditional cloud architectures, due to their centralized model. In these architectures, there is a main node that processes a huge amount of traffic coming from multiple connected systems. This model runs into a series of disadvantages that Edge Computing pretends to tackle.

The first problem is the increase of latency. IoT applications might require really low latency. Otherwise, the QoS (Quality-of-Service) and the QoE (Quality-of-Experience) might be downgraded. Besides, the distance between devices and datacenters is long enough to harm communications.

The second problem is the bandwidth consumption. In some contexts, the volume of generated data by the IoT devices can reach the size of gigabytes in seconds. So computational tasks can be extremely limited, especially if they are required to be executed in real time. Moreover, there is not only a problem related to the computational capabilities; the large amount of data causes network bottleneck.

The third problem is related to security. Every single data sent to the cloud is exposed to cyberattacks. Cloud services providers need to be continuously evolving and improving their security solutions, because an IoT system can be deployed in several and very different environments, such as industry and home automation. In addition, the capability of a system to be fault tolerant can be considered as a security feature. Several authors consider that, the more centralized a system is, less fault tolerant is.

Edge Computing emerges as a paradigm that attempts to improve the global efficiency of IoT deployments.

In order to reach this goal, Edge Computing allows the performance of computational, storage and networking tasks on the edge, that can be defined as a location between data source and datacenters. The gateway must be deployed at any part along this path, and must have enough computing capacity to perform this tasks, among the traditional ones such as data forwarding.

Consequently, data generated by sensors remain on the edge, where it is stored and processed; only valuable data is sent to the cloud. By reducing this amount of data, bottlenecks in cloud communications are reduced, so as the latency and bandwidth consumption.

Until now, cloud architectures feature techniques such as offloading, that consists on delegating to the cloud computational tasks that could not be performed locally due to lack of computing power; and caching, that enables quicker access to data which is frequently requested, so the system efficiency grows. These techniques can be also applied on edge devices.

These features enable potential new services and unexplored business models. Industries such as home automation or self-driving cars are widely benefited thanks to this technology.

Focusing on home automation, Edge Computing advantages can be easily appreciated.

The size of the data generated by sensors is limited; for instance, a temperature or humidity value. However, the number of sensors deployed in a house and the frequency with which these sensors make measurements considerably increase the amount of data generated

per day.

This data provides information that might be helpful in order to detect anomalies or risk situations, but, is it necessary to send every single data to the cloud?

In an Edge Computing architecture, statistical inference can be performed on the edge gateway, so the amount of data sent to the cloud is reduced. For example, the only value sent to the cloud is the outcome of the data inference, which might be a single number.

Therefore, there are less communications with the cloud, so bottleneck and cyberattacks exposures are reduced.

7.2. Objectives

The main objective of this project is to prove that it is possible to deploy an Edge Computing system on a low performance gateway. This system will add an additional intelligence layer over the capabilities that a traditional gateway performs: networking and forwarding tasks.

The solution will be named Edge Computing Manager (ECM).

This architecture must provide the capacity of performing data analysis on the data arriving from the sensors. It will be developed a software platform able to automate every task needed to compose an Edge Computing system: data ingestion, analysis and results publication.

Every component of the environment must fullfill several requirements regarding to functionality, processing capacity and security.

The edge gateway software will start on-demand services with several functionalities: data ingestion and data analysis.

It will be proved that the system can bear with high workloads in which several clients might take part, by asking for raw and processed data coming from different sources.

It will be developed an algorithm that manages the system resources, allowing dynamic allocation based on demand: in low demand scenarios, services will be allocated more resour-

ces but, the more services are requested, the less resources every service will be allocated.

It will be proved that the system is secure against several vulnerabilities. On the one hand, data privacy will be assured. Data owners will be the only user able to access to their data, due to the isolation of the services. On the other hand, mechanisms will be developed in order to prevent phishing attacks.

Performance analysis will prove the system, and the gateway, are able to listen to multiple services requests from several users and perform the requested tasks.

In order to contextualize the project, the gateway functionality fits the home automation scenario. Several sensors deployed along a house will send measures taken from environment conditions.

The idea is to perform data inference on the gateway, and the results to be sent to the cloud. A software platform will be deployed within the device, based on Docker containers. Some services will listen to sensors connections and ingest the data, while others will perform data analysis tasks.

This architecture might be included in a larger one, as figure 7.1 illustrates. Each apartment from a building has its own gateway, and each one connects with a common and more powerful device, which performs another inference and sends the result to another bigger gateway, which processes the data from every block from a neighbourhood. On the top of the hierarchy, it can be found a device that connects to the cloud.

7.3. Work methodology

The project development follows these steps:

- Edge gateway choice. The criterion follows two premises: low power consumption and enough computational capacity.
- First performance tests to prove the selected device is powerful enough to become the edge gateway. Due to its 32 bit architecture, check Docker and data analysis libraries

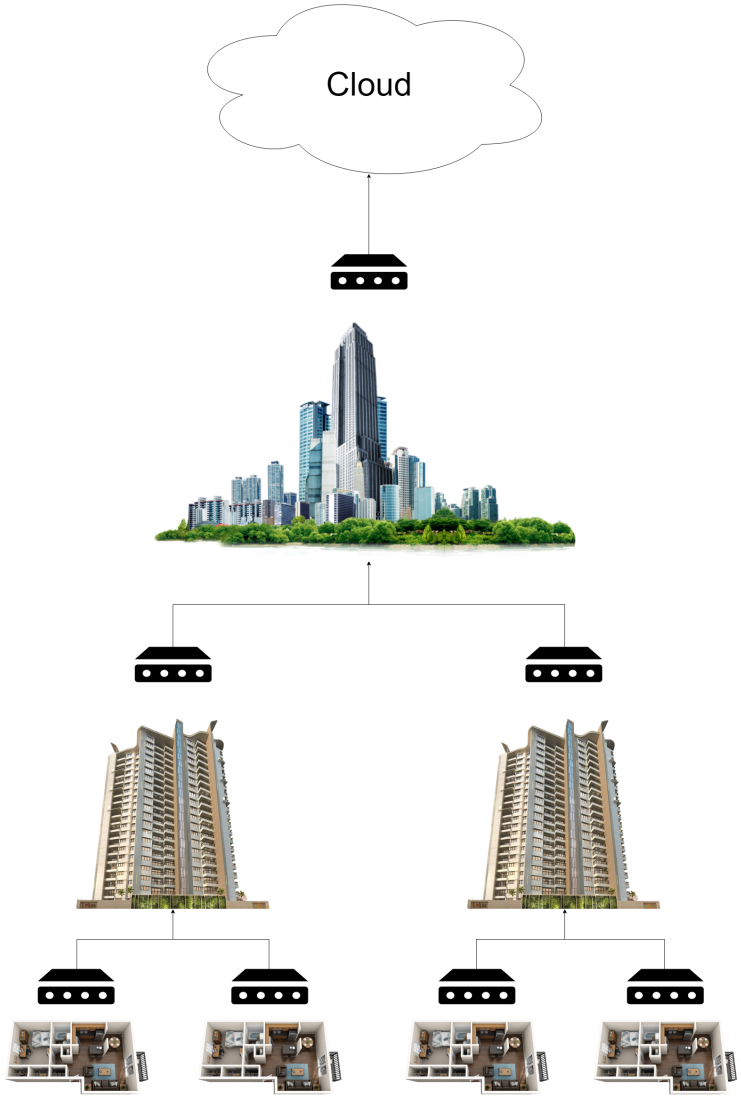


Figura 7.1: Target scenario architecture

can be installed.

- Requirements definition to illustrate system functionality: resources and containers management, clients connections, services to provide, etc.
- Solution development. Develop simple services that will gain complexity as the services are totally defined.

- Performance testing to prove that the goals are reached and to measure system efficiency.

7.4. Document organization

The document is composed by eight chapters explained as follows:

- First chapter is the introduction, that presents the project idea and the goals to reach.
- Second chapter introduces state of art. There will be listed several Edge Computing projects, initiatives and platforms. Potential edge devices and technologies will be also described.
- Third chapter describes the platform functionality. There will be shown ECM services, how the client interacts with them an which is the expected outcome from each of them.
- Fourth chapter illustrates the platform development with UML diagrams. System will be represented with deployment and object diagrams, and the functioning of each service with sequence diagrams.
- Fifth chapter shows the outcome from the performance tests, illustrated with graphics and tables. There will be also included some conclusions.
- Sixth chapter contains the general conclusions of the project.
- Seventh and eighth chapters are the translation of introduction and conclusions.

Capítulo 8

Conclusions

The main goal of the project was to prove that a low performance device could be used as an edge gateway within an Edge Computing architecture.

This document has shown the development and construction of an architecture inspired in the home automation industry. In this architecture, several sensors send the data they measure to an edge gateway that performances inference tasks.

The ECM platform has been built to work as an on-demand service provider, where clients request tasks to be performed.

Another goal of the project was to automate the global performance of the system. This automatic operation requested the following needs:

- Service providers management.
- Machine resources management, in order to avoid CPU overload.
- Clients management.

Every service must perform with enough computational capacity and efficiency, so that the time spent between the service is requested and the container starts working is minimal. Data inference tasks should also run quickly enough.

Besides, services must be isolated, so client's data privacy can be assure.

Therefore, it can be concluded that:

- **Edge gateway computational capacity is high enough:** The device can perform data inference tasks for admissible data sizes. The target environment of this project, where data is environmental conditions values, device's capacity allows inference tasks.

- **Limited number of services can run concurrently:** Despite the device is powerful enough to perform data inference, even when the amount of data generated is high, number of services that can be executed simultaneously is limited. This restriction exists because of the limited number of cores the device has and, when a container runs under a low core assignment, its performance is poor.

- **Platform is robust under high workloads:** When there are much service requests, and most of them are data inference tasks, system is able to perform the requested services. Both the resources management algorithm and Docker provide functionalities to make the system robust.

- **Platform operation is autonomous:** Containers from ECM platform run automatically since the client requests the service. Resources management is also autonomous, with no need of human intervention. It is able to allocate resources dynamically.

Bibliografía

[1] *AWS IoT*. URL: https://aws.amazon.com/es/greengrass/. (accessed: 26.03.2019).

[2] *Azure IoT Edge Website*. URL: https://azure.microsoft.com/es-es/services/iot-edge/. (accessed: 25.03.2019).

[3] Flavio Bonomi y col. "Fog computing and its role in the internet of things". En: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM. 2012, págs. 13-16.

[4] Abderrahmane Boudi y col. "Assessing Lightweight Virtualization for Security-as-a-Service at the Network Edge". En: *IEICE Transactions on Communications* (2018).

[5] Patrik Cerwall y col. "Ericsson mobility report". En: *Ericsson* (2015).

[6] Asaf Cidon y col. "Dynacache: dynamic cloud caching". En: *7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. 2015.

[7] *Docker*. URL: https://www.docker.com/. (accesed: 04.08.2019).

[8] *Docker Hub*. URL: https://hub.docker.com/. (accesed: 07.08.2019).

[9] *Docker overview*. URL: https://docs.docker.com/engine/docker-overview/. (accesed: 09.08.2019).

[10] Koustabh Dolui y Csaba Kiraly. "Towards multi-container deployment on IoT gateways". En: *arXiv preprint arXiv:1810.07753* (2018).

[11] *Edge Computing Manager Docker Hub*. URL: https://hub.docker.com/u/franpach. (accesed: 04.09.2019).

[12] *Edge Computing Manager GitHub*. URL: https://github.com/franpach/ECM. (accesed: 04.09.2019).

[13] *El Edge Computing como complemento para las arquitecturas cloud*. URL: https://www.lainnovacionnecesaria.com/el-edge-computing-como-complemento-para-las-infraestructuras-cloud/. (accesed: 29.08.2019).

[14] *Elijah: Cloudlet-based Edge Computing*. URL: http://elijah.cs.cmu.edu/. (accessed: 12.03.2019).

[15] *elswork/tensorflow-diy*. URL: https://hub.docker.com/r/elswork/tensorflow-diy. (accesed: 26.08.2019).

[16] Cheol-Ho Hong y Blesson Varghese. "Resource Management in Fog/Edge Computing: A Survey". En: *arXiv preprint arXiv:1810.00305* (2018).

[17] Yun Chao Hu y col. "Mobile edge computing—A key technology towards 5G". En: *ETSI white paper* 11.11 (2015), págs. 1-16.

[18] Bukhary Ikhwan Ismail y col. "Evaluation of docker as edge computing platform". En: *2015 IEEE Conference on Open Systems (ICOS)*. IEEE. 2015, págs. 130-135.

[19] Roberto Morabito. "Virtualization on internet of things edge devices with container technologies: a performance evaluation". En: *IEEE Access* 5 (2017), págs. 8835-8850.

[20] Roberto Morabito y Nicklas Beijar. "Enabling data processing at the network edge through lightweight virtualization technologies". En: *2016 IEEE International Conference on Sensing, Communication and Networking (SECON Workshops)*. IEEE. 2016, págs. 1-6.

[21] Roberto Morabito y col. "Consolidate IoT edge computing with lightweight virtualization". En: *IEEE Network* 32.1 (2018), págs. 102-111.

[22] Jianli Pan y James McElhannon. "Future edge cloud and edge computing for internet of things applications". En: *IEEE Internet of Things Journal* 5.1 (2018), págs. 439-449.

[23] Alex Reznik y col. "Developing software for multi-access edge computing". En: *ETSI White Paper* 20 (2017).

[24] Mahadev Satyanarayanan. "A brief history of cloud offload: A personal journey from odyssey through cyber foraging to cloudlets". En: *GetMobile: Mobile Computing and Communications* 18.4 (2015), págs. 19-23.

[25] Mahadev Satyanarayanan y col. "The case for vm-based cloudlets in mobile computing". En: *IEEE pervasive Computing* 4 (2009), págs. 14-23.

[26] Weisong Shi y col. "Edge computing: Vision and challenges". En: *IEEE Internet of Things Journal* 3.5 (2016), págs. 637-646.

[27] *Tensorflow*. URL: https://www.tensorflow.org/. (accesed: 06.08.2019).

[28] *Train your first neuronal network: basic classification*. URL: https://www.tensorflow.org/tutorials/keras/basic_classification. (accesed: 03.09.2019).

[29] *What is a Container?* URL: https://www.docker.com/resources/what-container. (accesed: 04.08.2019).